

AFRL-IF-RS-TR-2005-151
Final Technical Report
April 2005



A PARAMETRIC MODEL FOR LARGE SCALE AGENT SYSTEMS

University of Illinois at Urbana-Champaign

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K545

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-151 has been reviewed and is approved for publication

APPROVED: /s/

JAMES M. NAGY
Project Engineer

FOR THE DIRECTOR: /s/

JOSEPH CAMERA, Chief
Information & Intelligence Exploitation Division
Information Directorate

REPORT DOCUMENTATION PAGE

*Form Approved
OMB No. 074-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE APRIL 2005	3. REPORT TYPE AND DATES COVERED Final Jun 00 – Jan 05	
4. TITLE AND SUBTITLE A PARAMETRIC MODEL FOR LARGE SCALE AGENT SYSTEMS			5. FUNDING NUMBERS C - F30602-00-2-0586 PE - 62301E PR - TASK TA - 00 WU - 08
6. AUTHOR(S) Gul Agha			8. PERFORMING ORGANIZATION REPORT NUMBER N/A
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Illinois at Urbana-Champaign 109 Coble Hall 801 South Wright Street Champaign Illinois 61829-6200			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2005-151
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFED 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			
11. SUPPLEMENTARY NOTES AFRL Project Engineer: James M. Nagy/IFED/(315) 330-3173/ James.Nagy@rl.af.mil			
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			12b. DISTRIBUTION CODE
13. ABSTRACT (<i>Maximum 200 Words</i>) The goal of this project was to develop new techniques for the construction, description, and analysis of multi-agent systems. The characteristics of systems being addressed include a high degree of non-determinism (resulting from a large number of interactions) and unpredictability of the environment in which the agents operate. Specifically, we implemented tools for building robust and dependable large-scale multi-agent systems and studied methods for predicting and analyzing the behaviors of such systems. The project developed coordination methods for systems consisting of large numbers of agents.			
14. SUBJECT TERMS Autonomous Agent, MAS, Multi-Agent System, Large Scale Agent System, Coordination Model, Cooperation, Actor Framework, Formal Theory, Stochastic, Cellular Automata, Auctioning Scheme, Dynamic Environment, Distributed Task			15. NUMBER OF PAGES 422
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Table of Contents

1.	Objective.....	1
2.	Approach.....	2
3.	Accomplishments.....	4
3.1	Formal Analysis of Agent Systems.....	4
3.1.1	Reasoning about Agent Specifications using Probabilistic Rewrite Theory.....	4
3.1.2	Monitoring and Verification of Deployed Agent Systems.....	5
3.2	Multi-agent Modeling.....	6
3.2.1	The Constraint Optimization Framework.....	6
3.2.2	The Dynamic Distributed Task Assignment Framework.....	7
3.2.3	Cellular Automata-based Modeling.....	7
3.3	Coordination Framework: the Dynamic Forward/Reverse Auctioning Scheme.....	8
3.4	Large-Scale Multi-Agent Simulation: the Adaptive Actor	9
3.4.1	Adaptive Agent Distribution.....	9
3.4.2	Application Agent-Oriented Middle Agent Services.....	9
3.4.3	Message Passing for Mobile Agents.....	10
3.4.4	AAA for UAV Simulation.....	10
3.4.5	Experimental Results.....	11
3.5	Hardware Realization: The August 2004 Demo.....	14
4.	Publications.....	16
4.1	2005.....	16
4.2	2004.....	17
4.3	2003.....	20
4.4	2002.....	21
	Appendix A: Scalable Agent Distribution Mechanisms for Large-Scale UAV Simulation.....	22
	Appendix B: Efficient Agent Communication in Multi-Agent Systems.....	28
	Appendix C: Online Efficient Predictive Safety Analysis of Multi-Threaded.....	46
	Appendix D: An Instrumentation Technique for Online Analysis of Multi-Thread Program.....	60
	Appendix E: On Parallel vs. Sequential Threshold Cellular Automat.....	71
	Appendix F: Online Efficient Predictive Safety Analysis of Multithreaded Programs.....	91
	Appendix G: A Flexible Coordination Framework For Application Oriented Matchmaking and Brokering Services.....	106
	Appendix H: A Perspective on the Future of Massively Parallel Computing: Fine Grain vs. Coarse-Grain Parallel Models.....	130
	Appendix I: Concurrency vs. Sequential Interleavings in 1-D Threshold Cellular Automata.....	145
	Appendix J: An Instrumentation Techniques for Online Analysis of Multi-Threaded Programs.....	153

Appendix K: Efficient Decentralized Monitoring of Safety in Distributed Systems.....	161
Appendix L: On Efficient Communication and Service Agent Discovery in Multi-Agent Systems.....	171
Appendix M: On Specifying and Monitoring Epistemic Properties of Distributed Systems.....	178
Appendix N: Statistical and Monitoring Epistemic Properties of Distributed Systems.....	182
Appendix O: Maximal Clique Based Distributed Group Formations for Autonomous Agent.....	195
Appendix P: ATSpace: A Middle Agent to Support Application Oriented Matchmaking & Brokering Services.....	203
Appendix Q: Learning Continuous Time Markov Chains from Sample Executions.....	207
Appendix R: Towards Hierarchical Taxonomy of Autonomous Agents.....	217
Appendix S: Characterizing Configuration Spaces of Simple Threshold Cellular Automata.....	223
Appendix T: On Challenges on Modelling and Designing Resource Bounded Autonomous Agents Acting in Complex Dynamic Environments...233	
Appendix U: Some Modeling for Autonomous Agents Action Selection in Dynamic Partially Observable Environments.....	239
Appendix V: Task Assignment for a Physical Agent Team via a Dynamic Forward/Reverse Auction Mechanism.....	245
Appendix W: Dynamic Agent Allocation for Large-Scale Multi-Agent Applications.....	252
Appendix X: Maximal Clique Based Distributed Group Formation for Task Allocation in Large-Scale Multi-Agent Systems.....	267
Appendix Y: Generating Optimal Monitors for Extended Regular Expressions...282	
Appendix Z: Modeling A System of UAV's On a Mission.....	302
Appendix AA: Runtime Safety Analysis of Multi-threaded Programs.....	309
Appendix AB: Simple Genetic Algorithms for Pattern Learning the Role of Crossovers.....	319
Appendix AC: An Actor-Based Simulation for Studying UAC Coordination.....	323
Appendix AD: A Rewriting Based Model for Probabilistic Distributed Object Systems.....	332
Appendix AE: An Executable Specification of Asynchronous Pi-Calculus Semantics and May Testing in Maude 2.0.....	347
Appendix AF: Generating Optimal Linear Temporal Logic Monitors by Coinduction.....	368
Appendix AG: An Executable Specification of Asynchronous Pi-Calculus Semantics in May Testing in Maude 2.0.....	383
Appendix AH: Thin Middleware for Ubiquitous Computing.....	404

List of Figures

Figure 1: Inter-roles Interaction in the Forward/Reverse Auction Protocol.....	8
Figure 2: Three-Layered Architecture for UAV Simulations.....	11
Figure 3: ASC of UAVs for Performing a Given Mission.....	12
Figure 4: Runtimes for Static and Dynamic Agent Distribution.....	13
Figure 5: Experimental Environment.....	14

1 OBJECTIVE

The goal of this project was to develop new techniques for the construction, description, and analysis of multi-agent systems. The characteristics of systems being addressed include a high degree of non-determinism (resulting from a large number of interactions) and unpredictability of the environment in which the agents operate. Specifically, we implemented tools for building robust and dependable large-scale multi-agent systems and studied methods for predicting and analyzing the behaviors of such systems. The project developed coordination methods for systems consisting of large numbers of agents.

2 APPROACH

In order to effectively model large-scale systems, it is necessary to focus on properties of interest at a *macroscopic* level. Therefore, we used a divide and conquer approach in addressing the construction and analysis of large-scale agent systems and we modeled agent systems at two levels of abstractions: the *system level* and the *application level*.

Our goal at the system or implementation level was to ease the construction and analysis of *dependable* large-scale agent systems. At this level, we modeled agents as autonomous entities that interact via message passing schemes. Therefore, the actor framework provided a good starting point because it does not make any assumptions or stipulations about the logic of the application for which the actors are developed. The actor framework also has a rich formal theory that we extended and used to study high level specifications of agent systems. The formalisms we developed abstract irrelevant details and focus directly on the desired macroscopic properties. Furthermore, methods for modular and compositional specifications were considered so that simple specifications of specific properties or components can be composed to derive more complex specifications of larger systems.

Moreover, large-scale agent systems are inherently stochastic. The asynchrony and autonomy of widely distributed agents inevitably leads to non-determinism. This may be the case irrespective of the nature individual agent behaviors which might be deterministic or stochastic. The methods we developed accounted for this uncertainty by allowing stochastic descriptions of agent systems and providing techniques to reason about the likelihoods of possible evolutions of the systems.

In spite of the advantages of formal analysis, it has serious limitations when applied to large-scale agent systems. Typically, in order to keep the analysis tractable, one chooses only a few relevant abstractions to describe and reason about the system. While this is effective for certain purposes, formal analysis is not feasible for problems involving too many parameters. To address this problem, we developed the *Adaptive Actor Architecture* (AAA) which utilized various optimizing techniques to simulate large-scale agent system and derive empirical estimations of the desired properties.

We viewed our work at the system level as an enabling technique that facilitates the construction and testing of various multi-agent coordination models. These models focus *only* on the application logic at high levels of abstractions and are not concerned with the architecture used to implement the agents. Moreover, the work at the application level stimulated and motivated our work at the system level to cope with the scalability requirement of the application needs. Therefore, we used an evolutionary approach in which each level bootstrapped progress at the other one.

Our goal at the application or coordination level was to develop models and techniques that facilitate multi-agent teamwork or how to allow a group of agents to *cooperate* in order to

accomplish a high level team goal. To achieve this, we developed theoretical models that help us to understand the difficulty of multi-agent teamwork and provide a framework to parametrically analyze the tension between different aspects of the teamwork coordination problems. Specifically, we developed the *distributed constraint optimization framework* and we also used *cellular automata* to mathematically model multi-agent systems. We studied the role of agent autonomy in teamwork and developed natural epistemological and hierarchical taxonomy of different types of autonomous agents, strictly based on an agent's critical capabilities as seen by an outside observer.

While the above models were geared toward fostering our understanding of the teamwork problem, they were not suitable to serve as an executable model that enables each agent to decide about what action to do. Nevertheless, these efforts helped us to arrive at the proper level of abstraction to attack the teamwork problem. We represented the multi-agent teamwork problems as a *distributed task assignment problem* in which a group of agents is required to accomplish a set of tasks while maximizing a certain performance measures. We developed the *dynamic forward/reverse auctioning scheme* as a generic method for solving the distributed task assignment problem in dynamic environments.

To measure our progress at both the system and application levels, we used the TASK shared domain, *viz*, *Unmanned Aerial Vehicle* (UAV). Using our AAA, we developed large simulations agents modeling up to *10,000 UAVs* in a *surveillance* task, contrasting different coordination approaches for this task. We also modeled the problem of a team of UAVs in a search and rescue mission using our dynamic distributed task assignment framework and our dynamic auctioning scheme to derive agents' behavior in this domain. We evaluated this approach using physical agents, i.e., robots. The *robot-based simulation* consisted of up to 20 robots forming dynamic teams to carry out the surveillance task.

3 ACCOMPLISHMENTS

3.1 *Formal Analysis of Agent Systems*

In order to facilitate the analysis of the behavior of agent systems, it was necessary to develop techniques for specifying and reasoning about such systems. We have shown that these techniques can make use of three fundamental properties of agent systems:

- *Asynchrony*: Autonomous agents operate and communicate asynchronously.
- *Modularity*: Agent systems can be decomposed into concurrent components, each consisting of an ensemble of agents.
- *Locality and Non-interference*: Messages between agents are the only means of information flow, e.g., there are no shared variables.

We leveraged these properties in achieving significant progress in two main areas: *rewrite theory* and *distributed monitoring* of multi-agent systems.

3.1.1 Reasoning about Agent Specifications Using Probabilistic Rewrite Theory

We decided to extend the rewrite framework and use it for specification and reasoning about large-scale agent systems. The rewrite theory is an appealing formalism because of its support for abstraction by providing the ability to group several agent states into a single state, and several low level transitions into a big step transition. For example, a rule could abstractly state in a single step the transition of a group of agents from before a leader election procedure to after. Specifically, the final state of the group is specified without furnishing the details of how the transition is implemented. Therefore it results in a compact specification.

Nondeterminism exhibits itself in the system when two or more rules may be simultaneously applicable to a certain state. When such rules do not conflict they are allowed to proceed either concurrently or in an interleaving fashion. On the other hand, if the rules conflict, the system decides which rule to apply according to customizable *probabilistic* tactics. To enable such probabilistic tactics, we extended this rich formalism with probabilistic transitions which results in the Probabilistic Rewrite Theory formalism. We made significant progress in providing a precise formulation and semantic for this formalism.

We demonstrated how to use the *Probabilistic Rewrite Theory* to express systems with nondeterminism and probabilities, thus providing a way to reason about distributed systems, randomized algorithms, as well as systems where we have probabilistic models of communication delays, failures, etc. Moreover, we showed that Continuous Time Markov

Chains and Generalized Semi-Markov Processes can be naturally expressed in our rewriting model. We implemented a simulator for finitary probabilistic rewrite theories called *PMaude*. PMaude provides a tool to formally study agent systems, for example, to do performance modeling and studies of agent systems involving continuous variables.

3.1.2 Monitoring and Verification of Deployed Agent Systems

We used the actor properties (asynchrony, modularity, and locality) to develop a rich theory of agents. In particular, significant improvement can be made with respect to the efficiency of verification algorithms. We were able to show that instead of testing configurations under all possible execution environments -- an expensive and generally infeasible process because of the nondeterminism in agent systems -- it is possible to use an exponentially smaller set of traces which represents a canonical set of execution orders in such systems.

We also developed techniques for distributed monitoring of multi-agent systems. These techniques allow *automatic generation of local monitoring code* from a specification of a distributed property. The code is weaved with execution code to enable seamless monitoring of the specification. Finally, we developed methods for *statistical black-box testing* of probabilistic properties in a system

Finally, we developed techniques for the *scalable statistical analysis* of multi-agent systems. Specifically, we developed a new statistical approach for analyzing stochastic agent systems against specifications given in a sublogic of continuous stochastic logic (CSL). Unlike past numerical and statistical analysis methods, we assume that the system under investigation is an *unknown, deployed black-box* that can be passively observed to obtain sample traces, but cannot be controlled. Given a set of executions (obtained by Monte Carlo simulation) and a property, our techniques check, based on statistical hypothesis testing, whether the sample provides evidence to conclude the satisfaction or violation of a property, and computes a quantitative measure (*p*-value of the tests) of confidence in its answer; if the sample does not provide statistical evidence to conclude the satisfaction or violation of the property, the algorithm may respond with a “don’t know” answer.

3.2 Multi-agent Modeling

We have made significant progress in modeling multi-agent systems at the application level. Some of these models strive to provide a theoretical foundation to understand the difficulty of multi-agent coordination. They make use of rigorous mathematical optimization and models based on cellular-automata. Motivated by this analysis we also defined the dynamic distributed task assignment problem that enables us to use *economic models* to provide near *optimal solutions under resource constraints*.

3.2.1 The Constraint Optimization Framework

The goal of the constraint optimization framework is to mathematically provide a *parametric model* of a system of agents working as a team to accomplish a set of tasks while satisfying a heterogeneous set of physical and communication constraints. We used this framework to model a team of UAVs in a *surveillance* task. Specifically, we applied the framework to UAVs, where we model each UAV as an autonomous agent. Each agent has an individual utility and the goal of the system is to maximize a joint utility function. The application requires each UAV to plan its path in cooperation with other UAVs in order to accomplish an aerial survey of *dynamically evolving targets*. In this framework, we assumed the existence of a common clock and the ability of agents to communicate with each other in a given communication range by means of multi-cast messages. We also assumed that a certain number of geographical locations were of interest, and their value was dynamically changing over time.

Guided by the above formulation, and using our simulation environment (see Section 4), we simulated a number of agent strategies to understand the tension between variables in our parametric model. The simulation modeled UAV's intrinsic kinematics such as velocity and acceleration, constraints on each UAV's trajectory (e.g., collision avoidance), and constraints on resources such as fuel, available air corridors, and bounded communication radii. Each UAV's utility function was dependent on its position and the expected value of a target by the time the UAV would arrive there. In our model the value of the target declines with time. These set of simulations suggested two tentative conclusions:

There is an optimal number of UAVs for a given problem which can be established through our simulation engine. Beyond this number, more UAVs have a marginal *negative* impact on pay-offs. As we scale up, locality becomes more important than the utility of a target.

Moreover, we devised special algorithms to efficiently solve a certain instance of the constraint optimization problem were the goal is to form coalition among agents under the constraint of achieving *maximal cliques* in the underlying communication topology. This problem was directly motivated by multi-agent environments and applications where both node and link failures are to be expected, and where group and coalition robustness with respect to such failures is a highly desirable quality. These cliques, however, are restricted

to be of smaller than a given maximum ceiling size. This restriction allows computational tractability despite the fact that the general MAX-CLIQUE problem is NP-complete.

3.2.2 The Dynamic Distributed Task Assignment Framework

Using the *constraint optimization framework* we arrived at the proper level of abstraction to attack the teamwork problem. We realized, among other researchers albeit using a different formulation, that many multi-agent teamwork problems can be modeled as a *Dynamic Distrusted Task Assignment* (DDTA) problem. In the DDTA problem, a group of agents is required to accomplish a set of tasks while maximizing a certain performance measure. We identified that an effective solution to the DDTA problem needs to address two closely related questions:

1. How to find a near-optimal assignment from agents to tasks under resource constraints?
2. How to efficiently maintain the optimality of the assignment over time?

Guided by this formulation, we developed our solution to the DDTA problem, the Dynamic Forward/Reverse Auctioning scheme, described in Section 3. We applied this scheme to a UAV search and rescue scenario with promising results (see Section 5).

3.2.3 Cellular Automata-based Modeling

We made progress in developing a "starting point" mathematical model for agent systems using the classical *cellular automata* (CA). We intensely studied some extensions and modifications of the classical CA, so that the resulting graph or network automata are more faithful abstractions of a broad variety of large-scale multi-agent systems. We also developed natural epistemological and hierarchical taxonomy of different types of autonomous agents, strictly based on an agent's critical capabilities as seen by an outside observer.

3.3 Coordination Framework: The Dynamic Forward/Reverse Auctioning Scheme

The goal of the dynamic forward/reverse auction mechanism is to solve the dynamic distributed task assignment problem as posed in Section 2.2. A solution to the DDTA problem should ensure that task-agent assignment is always optimal with respect to the performance measure. Our approach can be characterized as a divide and conquer method that separately deals with combinatorial complexity and dynamicity – the two aspects of the DDTA problem. We addressed the first issue by extending an existing forward/reverse auction algorithm which was designed for bipartite maximal matching to find an initial near-optimal assignment. The extension makes it suitable for the distributed, asynchronous, multi-requirement aspects of the DDTA problem. However, the dynamicity of the environment compromises the optimality of the initial solution obtained via this modified algorithm. We address the dynamicity problem by using swapping to locally move agents between tasks. By linking these local swaps, the current assignment is morphed into one which is closer to what would have been obtained if we had re-executed the computationally more expensive auction algorithm.

We applied this dynamic auctioning scheme in the context of UAVs (Unmanned Aerial Vehicles) search and rescue mission and developed experiments using physical agents to show the feasibility of the proposed approach in the TASK August demo which featured twenty robots (targets and pursuers). Besides the experimental results, we conducted a theoretical analysis about the performance of swapping.

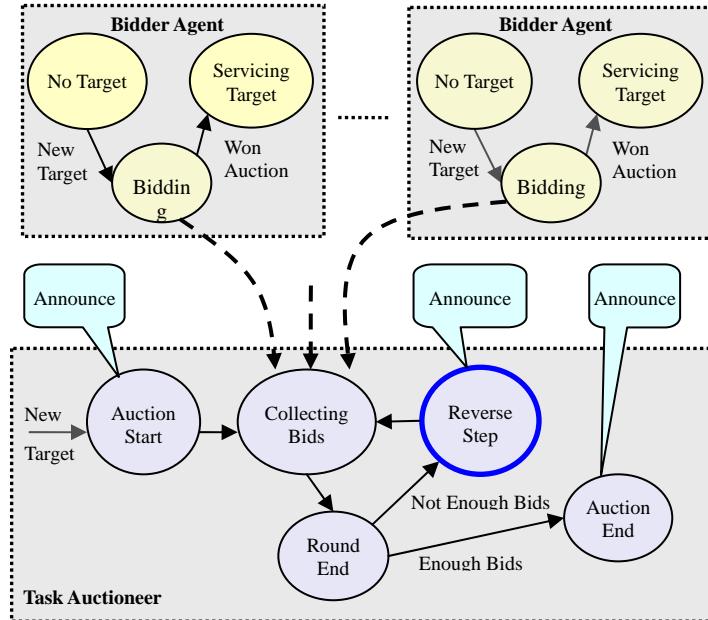


Figure 1: Inter-roles Interaction in the Forward/Reverse Auction Protocol

3.4 Large-scale Multi-agent Simulation: The Adaptive Actor Architecture

The *Adaptive Actor Architecture* (AAA) was one of the major accomplishments of this TASK project. The AAA is designed to support the construction of large-scale multi-agent applications by exploiting distributed computing techniques to efficiently distribute agents across a *distributed network* of computers. Distributing agents across nodes introduces inter-node communication that might eliminate any improvement in the runtime of large-scale multi-agent applications. The AAA uses several *optimizing techniques* to address three fundamental problems related to *agent communication* between nodes : *agent distribution*, *service agent discovery* and *message passing for mobile agents*. We will first discuss how these problems have been addressed in the AAA and then describe how the AAA was used in the UAV domain.

3.4.1 Adaptive Agent Distribution

Unless an agent requires some specific devices or services belonging to a certain computer node, the location of an agent does not affect the result of computation. However, the performance of agent applications may vary considerably according to the distribution pattern of agents, because agent distribution changes the inter-node communication pattern of agents, and the amount of inter-node communication considerably affects the overall performance. Therefore, if we could co-locate agents that intensively communicate with each other, the communication cost among agents could be minimized. Moreover, since the communication pattern of agents is continuously changing, agent distribution should be adaptive and dynamic .

If agents are dynamically distributed according to only their communication localities, some computer nodes could be overloaded with too many agents. Therefore, we distributed agents according to both their communication localities and the workload of computer nodes. The novel features of our approach are that this mechanism is based on the *communication locality of agent groups* as well as individual agents, and that the negotiation between computer nodes for agent migration occurs at the agent group level, but not at the individual agent level .

3.4.2 Application Agent-oriented Middle Agent Services

In *open* multi-agent systems where agents can enter and leave at any time, middle agent services such as brokering and matchmaking are very effective at finding service agents. Since *brokering* services can remove one message that may be very large , they may be more efficient than *matchmaking* services. However, because of the difficulty in expressing all search algorithms to a middle agent, in some cases we must use a matchmaking service

instead of brokering service. Previous solutions modify the search mechanism of the middle agent. But any change to the search mechanism in a middle agent affects other agents .

To handle the different interests of agents, we developed and implemented an *active interaction model for middle agent services*. In these services , the middle agent manages data, and search algorithms are given by application agents. With this separation of search algorithms from data, the middle agent can support the different interests of application agents at the same time without affecting other agents. Although application agents are located on different computer nodes, because their search algorithms are executed on the same computer node where data exist, the search algorithm can be performed efficiently, and the delivery overhead of search algorithms can be compensated with the performance benefit .

3.4.3 Message Passing for Mobile Agents

Message passing is the most fundamental service in multi-agent frameworks. Regardless of the locations of receiver agents, agent frameworks should provide reliable message passing. With dynamic agent distribution, agents may often change their locations . Sending a message to a *mobile agent* that has moved from its original computer node may require more than one message hop. If the sending node can directly deliver the message to the mobile agent, it will reduce communication time .

For this purpose, we developed a location-based message passing mechanism and a delayed message passing mechanism. The location-based message passing mechanism uses location information in the name of a receiver agent, and the name of an agent is updated by its current computer node whenever the agent changes its location. The delayed message passing mechanism allows a computer node to hold messages for a moving agent until the agent finishes its migration .

3.4.4 AAA for UAV Simulation

We used the AAA to build *ActorSim* with which we used to conduct several simulation runs to compare different coordination strategies for a UAV surveillance task. This simulation package provides simulation time management, environment-based agent interaction, kinematics of UAVs and targets, etc. We used this simulator to study the effectiveness of applying a number of multi-agent coordination mechanisms to the problem of cooperative surveillance in scenarios of up to 10,000 agents (5,000 UAVs and 5,000 targets). Moreover, we have designed a graphical simulation viewer for *ActorSim* in OpenGL. Good visualization is important not only for the spectators, but also for the designers of the higher-level system capabilities, such as, the agent capabilities of effective collaborative coordination and collision avoidance.

We have transferred a preliminary version of the *ActorSim* simulator and UAV simulation package to the Information Director of Air Force Research Laboratory (AFRL/IF) in Rome, NY. Rome Labs plans to customize the simulation toolkit for use by research teams at AFRL/IF.

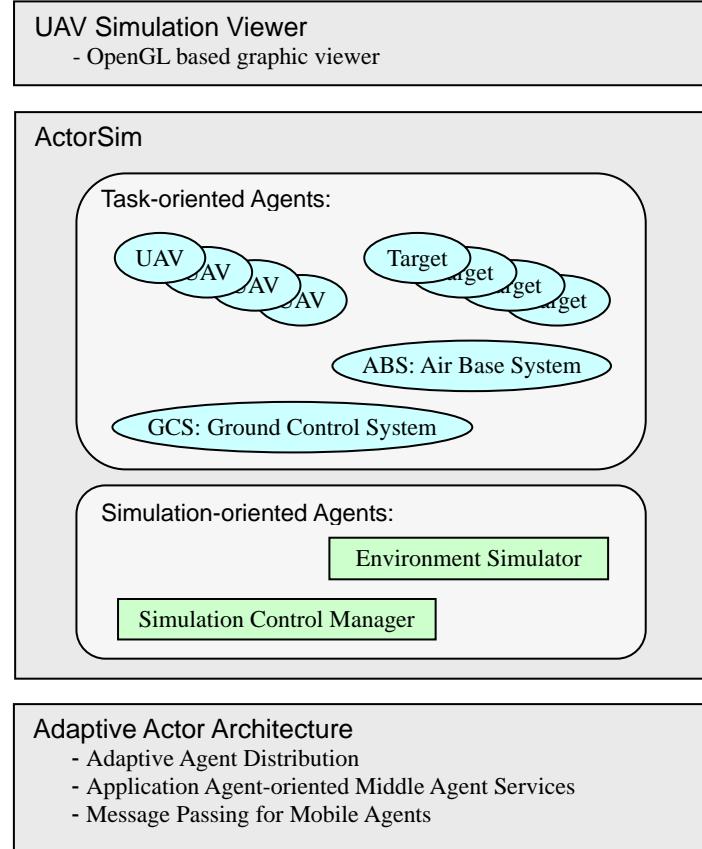


Figure 2: Three-Layered Architecture for UAV Simulations

3.4.5 Experimental Results

To investigate how a cooperation strategy influences the performance of a joint mission, we use *Average Service Cost* (ASC) as our metric. ASC is interpreted as additional navigation time to serve given targets, and is defined as follows:

$$ASC = \frac{\sum_i^n (NT_i - MNT)}{n}$$

where n is the number of UAVs, NT_i means navigation time of UAV i , MNT (Minimum Navigation Time) means average navigation time of all UAVs required for a mission when there are no targets.

Figure 3 depicts ASC for a team-based coordination strategy and a self-interest strategy. When the number of UAVs is increased, ASC is decreased in every case. This result explains that communication of UAVs is useful to handle targets, even though UAVs in the self-interest UAV strategy consumes quickly the value of a target when they handle the target together.

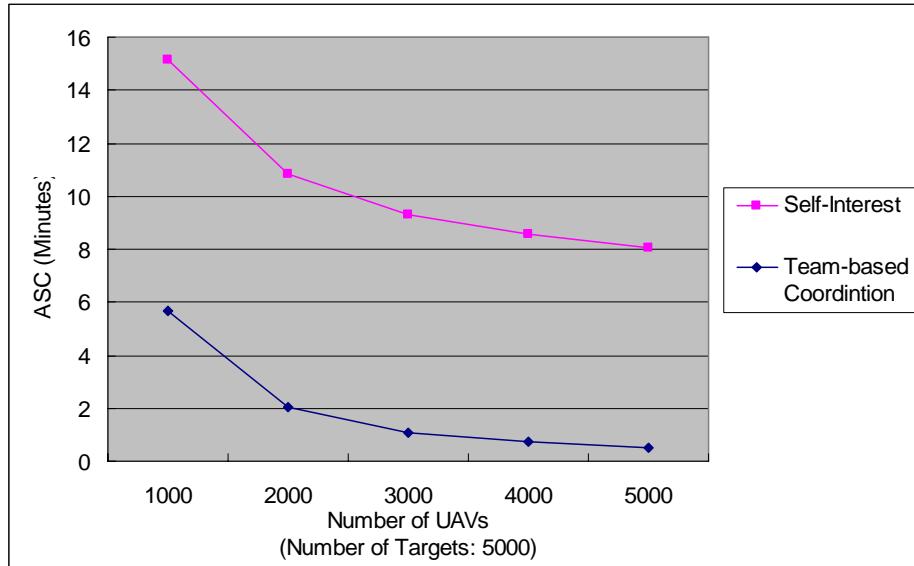


Figure 3: ASC of UAVs for Performing a Given Mission

To evaluate the potential benefit of adaptive agent distribution of AAA, we conducted the same simulations with two different agent distribution strategies: dynamic agent distribution and static agent distribution. Figure 4 depicts the difference of runtimes of simulations in two cases. Even though the dynamic agent distribution in our simulations includes the overhead for monitoring and decision making, the overall performance of dynamic agent distribution overwhelms that of static agent distribution. As the number of agents is increased, the ratio also generally increases. With 10,000 agents, the simulation using the dynamic agent allocation is more than five times faster than the simulation with a static agent allocation.

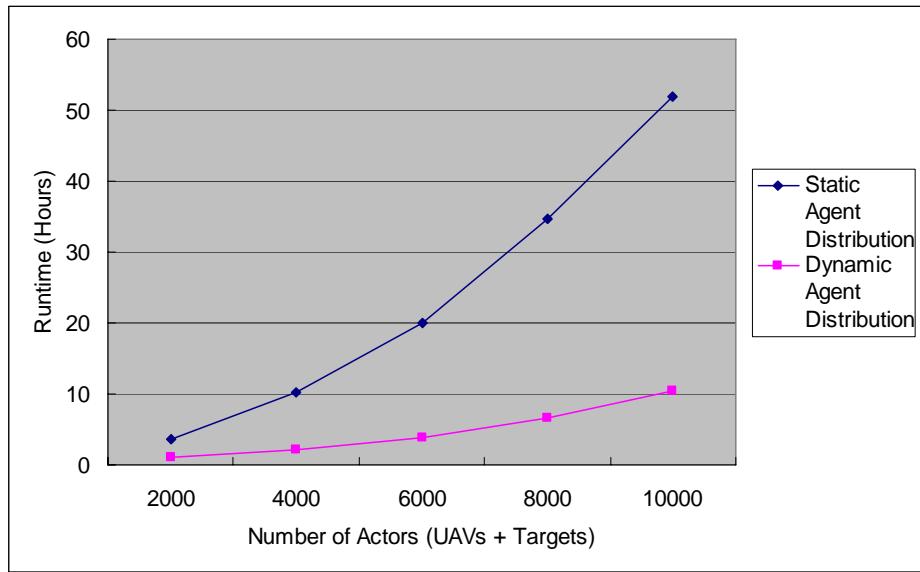


Figure 4: Runtimes for Static and Dynamic Agent Distribution

3.5 Hardware Realization: The August 2004 Demo

The goal of our August hardware demo was to demonstrate our techniques in a realistic environment. For this purpose, we created a search and rescue mission. In this domain, a collection of UAVs roam a rectangle mission area looking for targets (downed pilots, injured civilians, etc.). These targets move according to a pre-determined path not known to the UAVs. Each target has a step utility function and requires a minimum number of UAVs to be serviced. This step utility function means that before the target gets its required number of UAVs, none of its utility can be consumed by the team. Once a requisite number of UAVs arrive near the target, it is deemed to have been serviced. UAVs monitor targets and coordinate the groups that service them subject to maximizing the total team benefit. We have modeled the above scenario using the DDTA formulation as mentioned in Section 2.2 and applied our dynamic forward/reverse auction mechanism to derive agent behaviors in this domain.

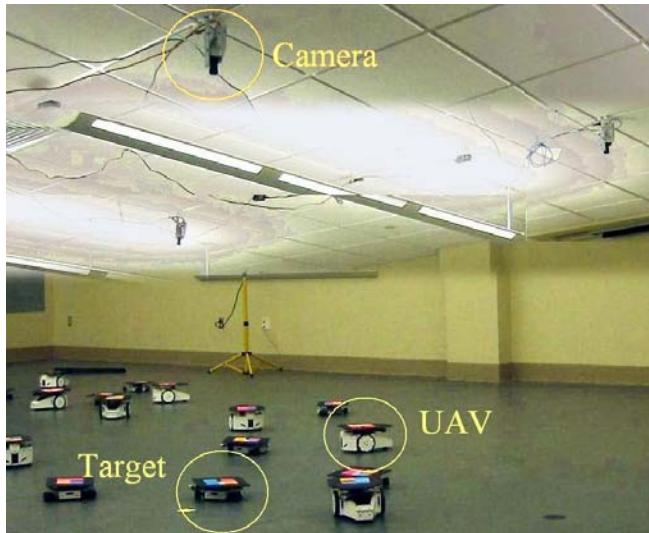


Figure 5: Experimental Environment

We modeled UAVs and targets as *robot cars*. Each car was controlled by an iPAQ PDA running Microsoft Pocket PC and receives localization information from a leader vision server collaborating data from four vision servers, each of which is connected to an overhead video camera. A vision server takes images from a camera, searches for unique color plates mounted on each robot car, and calculates the corresponding robot's identification and heading. A leader vision server takes *localization information* from each vision server, and sends filtered and regulated localization information to the iPAQs. The iPAQs use an internal WiFi interface for inter-agent communication. Different cars are used to represent UAVs and targets. It is quite clear that this hardware setting makes discovering logical errors in the software implementation and tracing the agents' behavior very hard. To deal with these issues, we developed a hardware/software shared agent code

architecture that allows us to simulate this hardware setting in software while at the same time guaranteeing interoperability when porting this code to the hardware setting. The main design philosophy of the system was to ease parallel development and testing of the code. The agent's (UAV/Target) implementation is isolated from the architecture on which the system is running. The system was developed to run in one of two modes: a *simulated mode* or a *real node*. This architecture helped us accelerate the development cycle when preparing for the hardware final demo.

Another technical issue we faced during this hardware demo, which was not apparent in software simulations, was collision avoidance. While collisions can be abstracted away in a large-scale software simulation of a multi-agent system, the issue is critical in the context of real vehicles. With up to twenty robots in our hardware demo, operating in a relatively small area ($8\text{ m} \times 6\text{ m}$), collision avoidance poses significant challenges. To address this problem, we developed *online path planning* and re-planning heuristics for collision avoidance as well as group coordination behavior when pursuing the assigned target. All of these techniques were demonstrated successfully during a demonstration for DARPA in August 2004.

4 PUBLICATIONS

4.1 2005

Myeong-Wuk Jang and Gul Agha, "Scalable Agent Distribution Mechanisms for Large-Scale UAV Simulations," *The International Conference of Integration of Knowledge Intensive Multi-Agent Systems KIMAS '05, Scalable Agents Session*, Waltham, Massachusetts, April 18-21, 2005.

Myeong-Wuk Jang, Amr Ahmed and Gul Agha, "Efficient Communication in Multi-Agent Systems," *LNCS Special Issue on Software Engineering for Large Scale Multi-Agent Systems*, to be published 2005.

Koushik Sen, Grigore Rosu and Gul Agha, "Online Efficient Predictive Safety Analysis of Multithreaded Programs", *International Journal on Software Technology and Tools Transfer STTT*, to appear 2005.

Grigore Rosu and Koushik Sen, "An Instrumentation Technique for Online Analysis of Multithreaded Programs", *Special Issue of Concurrency and Computation: Practice and Experience (CC:PE)*, to appear 2005.

Predrag Tomic and Gul Agha, "On Parallel vs. Sequential Threshold Cellular Automata," *Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign*, to appear 2005.

4.2 2004

Koushik Sen, Grigore Rosu and Gul Agha, "Online Efficient Predictive Safety Analysis of Multithreaded Programs," *In Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS '04*, Springer-Verlag Lecture Notes in Computer Science, volume 2988, Barcelona, Spain, March 29-April 2, 2004, pages 123-138.

Myeong-Wuk Jang, Amr Ahmed and Gul Agha, "A Flexible Coordination Framework for Application-Oriented Matchmaking and Brokering Services," *Technical Report UIUCDCS-R-2004-2430, Department of Computer Science, University of Illinois at Urbana-Champaign*, April 2004.

Predrag Tomic, "A Perspective on the Future of Massively Parallel Computing: Fine Grain vs. Coarse-Grain Parallel Models," *Proceedings of the First ACM Conference on Computing Frontiers (CF '04)*, Ischia, Italy, April 14-16, 2004, pages 488-502.

Predrag Tomic and Gul Agha, "Concurrency vs. Sequential Interleavings in 1-D Threshold Cellular Automata," *Proceedings of The 18th International Parallel and Distributed Processing Symposium IPDPS '04, Advances in Parallel and Distributed Computing Models Workshop*, Santa Fe, New Mexico, USA, April 26-30, 2004, page 179b.

Grigore Rosu and Koushik Sen, "An Instrumentation Technique for Online Analysis of Multithreaded Programs," *Workshop on Parallel and Distributed Systems: Testing and Debugging PADTAD '04*, Santa Fe, New Mexico, USA, April 30, 2004, page 268.

Koushik Sen, Abhay Vardhan, Gul Agha and Grigore Rosu, "Efficient Decentralized Monitoring of Safety in Distributed Systems," *In Proceedings of 26th International Conference on Software Engineering ICSE '04*, Edinburgh, Scotland, United Kingdom, May 23-28, 2004, pages 418-427.

Myeong-Wuk Jang and Gul Agha, "On Efficient Communication and Service Agent Discovery in Multi-agent Systems," *Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS '04)*, Edinburgh, Scotland, May 24-25, 2004, pages 27-33.

Koushik Sen, Abhay Vardhan, Gul Agha and Grigore Rosu, "On Specifying and Monitoring Epistemic Properties of Distributed Systems," *Second International Workshop on Dynamic Analysis, WODA '04*, pages 32-35, Edinburgh, Scotland, United Kingdom, May 25, 2004, pages 32-35.

Koushik Sen, Mahesh Viswanathan and Gul Agha, "Statistical Model Checking of Black-Box Probabilistic Systems," *Proceedings from the 16th International Conference on Computer Aided Verification CAV '04*, Springer-Verlag, *Lecture Notes in Computer Science*, volume 3114, Boston, MA, USA, July 13-17, 2004, pages 202-215.

Predrag Totic and Gul Agha, "Maximal Clique Based Distributed Group Formation for Autonomous Agent Coalitions," *Third International Joint Conference on Agents & Multi Agent Systems AAMAS '04, Coalitions and Teams Workshop*, New York, New York, USA, July 19-23, 2004.

Myeong-Wuk Jang, Amr Ahmed and Gul Agha, "ATSpace: A Middle Agent to Support Application-Oriented Matchmaking and Brokering Services," *Proceedings of IEEE/WIC/ACM Intelligent Agent Technology 2004 (IAT '04)*, Beijing, China, September 20-24, 2004, pages 393-396.

Koushik Sen, Mahesh Viswanathan and Gul Agha, "Learning Continuous Time Markov Chains from Sample Executions," *First International Conference on Quantitative Evaluation of Systems QEST '04*, Enschede, The Netherlands, September 27-30, 2004, pages 146-155.

Predrag Totic and Gul Agha, "Towards a Hierarchical Taxonomy of Autonomous Agents," *Proceedings from the IEEE International Conference on Systems, Man and Cybernetics SMC '04*, The Hague, The Netherlands, October 10-13, 2004.

Predrag Totic and Gul Agha, "Characterizing Configuration Spaces of Simple Threshold Cellular Automata," *Sixth International Conference on Cellular Automata for Research and Industry*, Amsterdam, The Netherlands, October 25-27, 2004, Springer-Verlag, *Lecture Notes in Computer Science*, volume 3305, 2004, pages 861-870.

Predrag Totic and Gul Agha, "On Challenges in Modeling and Designing Resource-Bounded Autonomous Agents Acting in Complex Dynamic Environments," *Proceedings of the IASTED International Conference on Knowledge Sharing and Collaborative Engineering KSCE '04*, St. Thomas, US Virgin Islands, November 22-24, 2004.

Predrag Totic and Gul Agha, "Some Models for Autonomous Agents' Action Selection in Dynamic Partially Observable Environments," *Proceedings from the IASTED International Conference on Knowledge Sharing and Collaborative Engineering KSCE '04*, St. Thomas, US Virgin Islands, November 22-24, 2004.

Amr Ahmed, Abhilash Patel, Tom Brown, Myeong-Wuk Jang and Gul Agha, "Task Assignment for a Physical Agent Team via a Dynamic Forward/Reverse Auction Mechanism," *Technical Report UIUCDCS-R-2004-2507, Department of Computer Science, University of Illinois at Urbana-Champaign*, December 2004.

Myeong-Wuk Jang and Gul Agha, "Dynamic Agent Allocation for Large-Scale Multi-Agent Applications," *International Workshop on Massively Multi-Agent Systems*, Kyoto, Japan, December 10-11, 2004, pages 19-33.

Predrag Totic and Gul Agha, "Maximal Clique Based Distributed Group Formation for Task Allocation in Large-Scale Multi-Agent Systems," *Proceedings from the Workshop on Massively Multi-Agent Systems*, Kyoto, Japan, December 10-11, 2004.

4.3 2003

Koushik Sen and Grigore Rosu, "Generating Optimal Monitors for Extended Regular Expressions," *Proceedings of 3rd Workshop on Runtime Verification RV '03*, Elsevier Science *Electronic Notes in Theoretical Computer Science*, volume 89, issue 2, Boulder, Colorado, USA, July 13, 2003.

Predrag Tasic, Myeong-Wuk Jang, Smitha Reddy, Joshua Chia, Liping Chen and Gul Agha, "Modeling a System of UAVs on a Mission," *Proceedings of the 7th World Multiconference on Systemics, Cybernetics, and Informatics SCI '03*, July 27-30, 2003, pages 508-514.

Koushik Sen, Grigore Rosu and Gul Agha, "Runtime Safety Analysis of Multithreaded Programs," *Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering FSE/ESEC '03*, Helsinki, Finland, September 3-5, 2003, pages 337-346.

Predrag Tasic and Gul Agha, "Simple Genetic Algorithms for Pattern Learning: The Role of Crossovers," *5th International Workshop on Frontiers in Evolutionary Algorithms FEA '03 Proceedings from the 7th Joint Conference on Information Sciences*, Carey, North Carolina, USA, September 26-30, 2003.

Myeong-Wuk Jang, Smitha Reddy, Predrag Tasic, Liping Chen and Gul Agha, "An Actor-based Simulation for Studying UAV Coordination," *15th European Simulation Symposium ESS 2003*, Delft, The Netherlands, October 26-29, 2003, pages 593-601.

Nirman Kumar, Koushik Sen, Jose Meseguer and Gul Agha, "A Rewriting Based Model for Probabilistic Distributed Object Systems," *In Proceedings of 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems FMOODS '03*, Springer-Verlag *Lecture Notes in Computer Science*, volume 2884, Paris, France, November 19-21, 2003, pages 32-46.

Koushik Sen, Grigore Rosu and Gul Agha, "Generating Optimal Linear Temporal Logic Monitors by Coinduction," *In Proceedings of 8th Asian Computing Science Conference ASIAN '03*, Springer-Verlag *Lecture Notes in Computer Science*, volume 2896, Mumbai, India, December 10-12, 2003, pages 260-275.

Predrag Tasic and Gul Agha, "Understanding and Modeling Agent Autonomy in Dynamic Multi-Agent, Multi-Task Environments," *Proceedings of the First European Workshop on Multi-Agent Systems EUMAS '03*, Oxford, England, UK, December 18-19, 2003.

4.4 2002

Koushik Sen, Gul Agha in Dan C. Marinescu and Craig Lee, "Thin Middleware for Ubiquitous Computing," *Process Coordination and Ubiquitous Computing*, CRC Press, September 2002, pages 201-213.

Prasanna V. Thati, Koushik Sen and Narciso Marti Oliet, "An Executable Specification of Asynchronous Pi-calculus and May-testing in Maude 2.0," *International Workshop on Rewriting Logic and its Applications WRLA '02*, Elsevier Science *Electronic Notes in Theoretical Computer Science*, volume 71, Pisa, Italy, September 13-21, 2002.

Scalable Agent Distribution Mechanisms for Large-Scale UAV Simulations

Myeong-Wuk Jang and Gul Agha
 Department of Computer Science
 University of Illinois at Urbana-Champaign
 Urbana, IL 61801, USA
 {mjang, agha}@uiuc.edu

Abstract — A cluster of computers is required to execute large-scale multi-agent. However, such execution incurs an inter-node communication overhead because agents intensively communicate with other agents to achieve common goals. Although a number of dynamic load balancing mechanisms have been developed, these mechanisms are not scalable in multi-agent applications because of the overhead involved in analyzing the communication patterns of agents. This paper proposes two scalable dynamic agent distribution mechanisms; one mechanism aims at minimizing agent communication cost, and the other mechanism attempts to move agents from overloaded agent platforms to lightly loaded platforms. Our mechanisms are fully distributed algorithms and analyze only coarse-grain communication dependencies of agents, thus providing scalability. We describe the results of applying these mechanisms to large-scale micro UAV (Unmanned Aerial Vehicle) simulations involving up to 10,000 agents.

1. INTRODUCTION

As the number of agents in large-scale multi-agent applications increases by orders of magnitude (e.g. see [7, 9, 10]), distributed execution is required to improve the overall performance of applications. However, parallelizing the execution on a cluster may lead to inefficiency; a few computer nodes may be idle while others are overloaded. Many dynamic load balancing mechanisms have been developed to enable efficient parallelization [1, 3, 6]. However, these mechanisms may not be applicable to multi-agent applications because of the different computation and communication behavior of agents [4, 9].

Some load balancing mechanisms have been developed for multi-agent applications [4, 5], but these mechanisms require a significant overhead to gather information about the communication patterns of agents and analyze the

information. Therefore, we believe these mechanisms may not be scalable. In this paper, we propose two *scalable* agent distribution mechanisms; one mechanism aims at minimizing agent communication cost, and the other mechanism attempts to move agents from an overloaded agent platform to lightly loaded agent platforms. These two mechanisms are developed as fully distributed algorithms and analyze only coarse-grain communication dependencies of agents instead of their fine-grain communication dependencies.

Although the scalability of multi-agent systems is an important concern in the design and implementation of multi-agent platforms, we believe such scalability cannot be achieved without customizing agent platforms for a specific multi-agent application. In our agent systems, each computer node has one agent platform, which manages scheduling, communication, and other middleware services for agents executing on the computer node. Our multi-agent platform is adaptive to improve the scalability of the entire system. Specifically, large-scale micro UAV (Unmanned Aerial Vehicle) simulations involving up to 10,000 agents are studied using our agent distribution mechanisms.

The paper is organized as follows: Section 2 discusses the scalability issues of multi-agent systems. Section 3 describes two agent distribution mechanisms implemented in our agent platform. Section 4 explains our UAV simulations and their interaction with our agent distribution mechanisms. Section 5 shows the preliminary experimental results to evaluate the performance gain resulting from the use of these mechanisms. The last section concludes this paper with a discussion of our future work.

2. SCALABILITY OF MULTI-AGENT SYSTEMS

The scalability of multi-agent systems depends on the structure of an agent application as well as the multi-agent platform. For example, when a distributed multi-agent application includes centralized components, these components can become a bottleneck of parallel execution, and the application may not be scalable. Even when an application has no centralized components, agents may use middle agent services, such as brokering or matchmaking services, supported by agent platforms, and the agent platform-level component that supports these services may

become a bottleneck for the entire system.

The goal of executing a cluster of computers for a single multi-agent application is to improve performance by taking advantage of parallel execution. However, balancing the workload on computer nodes requires a significant overhead from gathering the global state information, analyzing the information, and transferring agents very often among computer nodes. When the number of computer nodes and/or that of agents are very large, achieving optimal load balancing is not feasible. Therefore, we use a load sharing approach which move agents from an overloaded computer node, but the workload balance between different computer nodes is not required to be optimal.

Another important factor in the performance of large-scale multi-agent applications is agent communication cost. This cost may significantly affect the performance of multi-agent systems, when agents distributed on separate computer nodes communicate intensively with each other. Even though the speed of local networks has considerably increased, the intra-node communication for message passing is much faster than inter-node communication. Therefore, if we can collocate agents which communicate intensively with each other, communication time may significantly decrease. Distributing agents statically by a programmer is not generally feasible, because the communication patterns among agents may change over time. Thus agents should be dynamically reallocated according to their communication localities of agents, and this procedure should be managed by a multi-agent platform.

Because of a large number of agents in a single multi-agent application, the overhead from gathering the communication patterns of agents and analyzing such patterns would significantly affect the overall performance of the entire system. For example, when there are n agents and unidirectional communication channels between agents are used, the maximum number of possible communication connections among agents is $n \times (n-1)$. If the communication patterns between agents and agent platforms are considered for dynamic agent distribution, the maximum number of communication connections becomes $n \times m$ where m is the number of agent platforms. Usually, m is much less than n .

Another important concern for the scalability is the location of agent distributor that performs dynamic agent distribution. If a centralized component handles this task, the communication between this component and agent platforms may be significantly increased and the component may be the bottleneck of the entire system. Therefore, when multi-agent systems are large-scale, more simplified information for decision making and distributed algorithms would be more applicable for the scalability of dynamic agent distribution mechanisms.

For the purpose of dynamic agent distribution, each agent platform may monitor the status of its computer node and the communication patterns of agents on it, and distribute

agents according to their communication localities and the workload of its computer node. However, with the interaction with multi-agent applications, the quality of this service may be improved. For example, multi-agent applications may initialize or change parameters of dynamic agent distribution during execution for the better performance of the entire system. For the interaction between agent applications and platforms, we use a *reflective mechanism* [11]; agents in applications are supported by agent platforms, and the services of agent platforms may be controlled by agents in applications. This paper shows how our multi-agent applications (e.g., UAV simulations) interact with our dynamic agent distribution mechanisms.

3. DYNAMIC AGENT DISTRIBUTION

This section describes two mechanisms for dynamic agent distribution: a *communication localization* mechanism collocates agents which communicate intensively with each other, and a *load sharing* mechanism moves agent groups from overloaded agent platforms to lightly loaded agent platforms. Although the purpose of these two mechanisms is different, the mechanisms consist of similar process phases and share the same components in an agent platform. Both these two mechanisms are also designed as fully distributed algorithms. Figure 1 shows the state transition diagram for these two agent distribution mechanisms.

For these dynamic agent distribution services, four system components in our agent platform are mainly used. A detailed explanation of system components is described in [9].

1. *Message Manager* takes charge of message passing between agents.

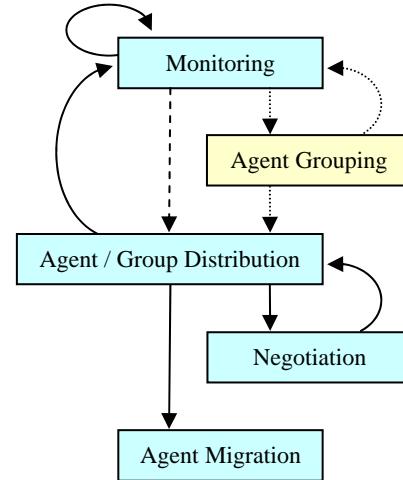


Figure 1 - State Transition Diagram for Dynamic Agent Distribution. The solid lines are used for both mechanisms, the dashed line is used only for the communication localization mechanism, and the dotted lines are used only for the load sharing mechanism.

2. *System Monitor* periodically checks the workload of its computer node.
3. *Actor Allocation Manager* is responsible for dynamic agent distribution.
4. *Actor Migration Manager* moves agents to other agent platforms.

3.1. Agent Distribution for Communication Locality

The communication localization mechanism handles the dynamic change of the communication patterns of agents. As time passes, the communication localities of agents may change according to the changes of agents' interests. By analyzing messages delivered between agents, agent platforms may decide what agent platform an agent should be located on. Because an agent platform can neither estimate the future communication patterns of agents nor know how agents on other platforms may migrate, local decision of an agent platform cannot be perfect. However, our experiments show that in case of our applications, reasonable performance can be achieved. The communication localization mechanism consists of four phases: *monitoring*, *agent distribution*, *negotiation*, and *agent migration* (see Figure 1).

Monitoring Phase — The Actor Allocation Manager checks the communication patterns of agents with the assistance from the Message Manager. Specifically, the Actor Allocation Manager uses information about both the sender agent and the agent platform of the receiver agent of each message. This information is maintained with a variable M representing all agent platforms communicating with each agent on the Manager's platform.

The Actor Allocation Manager periodically computes the communication dependencies $C_{ij}(t)$ at time t between agent i and agent platform j using equation 1.

$$C_{ij}(t) = \alpha \left(\frac{M_{ij}(t)}{\sum_k M_{ik}(t)} \right) + (1 - \alpha) C_{ij}(t-1) \quad (1)$$

where $M_{ij}(t)$ is the number of messages sent from agent i to agent platform j during the t -th time step, and α is a coefficient representing the relative importance between recent information and old information.

Agent Distribution Phase — After a certain number of repeated monitoring phases, the Actor Allocation Manager computes the communication dependency ratio of an agent between its current agent platform n and all other agent platforms, where the communication dependency ratio R_{ij} between agent i and platform j is defined using equation 2.

$$R_{ij} = \frac{C_{ij}}{C_{in}}, \quad j \neq n \quad (2)$$

When the maximum value of the communication

dependency ratio of an agent is larger than a predefined threshold θ , the Actor Allocation Manager assigns the agent to a *virtual agent group* that represents a remote agent platform.

$$k = \arg \max_j (R_{ij}) \quad \text{and} \quad R_{ik} > \theta \rightarrow a_i \in G_k \quad (3)$$

where a_i represents agent i , and G_k means virtual agent group k .

After the Actor Allocation Manager checks all agents, and assigns some of them to virtual agent groups, it starts the negotiation phase, and information about the communication dependencies of agents is reset.

Negotiation Phase — Before the agent platform P_1 moves the agents assigned to a given virtual agent group to destination agent platform P_2 , the Actor Allocation Manager of P_1 communicates with that of P_2 to check the current status of P_2 . Only if P_2 has enough space and the percentage of its CPU usage is not continuously high, the Actor Allocation Manager of P_2 accepts the request. Otherwise, the Manager of P_2 responds with the number of agents that it can accept. In this case, the P_1 moves only a subset of the virtual agent group.

Agent Migration Phase — Based on the response of a destination agent platform, the Actor Allocation Manager of the sender agent platform initiates migration of entire or part of agents in the selected virtual agent group. When the destination agent platform has accepted part of agents in the virtual agent group, agents to be moved are selected according to their communication dependency ratios. After the current operation of a selected agent finishes, the Actor Migration Manager moves the agent to its destination agent platform. After the agent is migrated, it carries out its remaining operations.

3.2. Agent Distribution for Load Sharing

The agent distribution mechanism for communication locality handles the dynamic change of the communication patterns of agents, but this mechanism may overload a platform once more agents are added to this platform. Therefore, we provide a load sharing mechanism to redistribute agents from overloaded agent platforms to lightly loaded agent platforms. When an agent platform is overloaded, the System Monitor detects this condition and activates the agent redistribution procedure. Since agents had been assigned to their current agent platforms according to their recent communication localities, choosing agents randomly for migration to lightly loaded agent platforms may result in cyclical migration. The moved agents may still have high communication rate with agents on their previous agent platform. Our load sharing mechanism consists of five phases: *monitoring*, *agent grouping*, *group distribution*, *negotiation*, and *agent migration* (see Figure 1).

Monitoring Phase — The System Monitor periodically

checks the state of its agent platform; the System Monitor gets information about the current processor usage and the memory usage of its computer node by accessing system call functions and maintains the number of agents on its agent platform. When the System Monitor decides that its agent platform is overloaded, it activates an agent distribution procedure. When the Actor Allocation Manager is notified by the System Monitor, it starts monitoring the local communication patterns of agents in order to partition them into *local agent groups*. For this purpose, an agent which was not previously assigned to an agent group is randomly assigned to some agent group.

To check the local communication patterns of agents, the Actor Allocation Manager uses information about the sender agent, the agent platform of the receiver agent, and the agent group of the receiver agent of each message. After a predetermined time interval the Actor Allocation Manager updates the communication dependencies between agents and local agent groups on the same agent platform using equation 1 using $c_{ij}(t)$ and $m_{ij}(t)$ instead of $C_{ij}(t)$ and $M_{ij}(t)$. In the modified equation 1, j represents a local agent group instead of an agent platform, $c_{ij}(t)$ represents the communication dependency between agent i and agent group j at the time step t , and $m_{ij}(t)$ is the number of messages sent from agent i to agents in local agent group j during the t -th time step. Note that in this case $\sum_k m_{ik}(t)$

represents the number of messages sent by the agent i to all agents in its current agent platform, and in general the value of the parameter α will be different.

Agent Grouping Phase — After a certain number of repeated monitoring phases, each agent i is re-assigned to a local agent group whose index is decided by $\arg \max_j(c_{ij}(t))$. Since the initial group assignment of agents

may not be well organized, the monitoring and agent grouping phases are repeated several times. After each agent grouping phase, information about the communication dependencies of agents is reset.

Group Distribution Phase — After a certain number of repeated monitoring and agent grouping phases, the Actor Allocation Manager makes a decision to move an agent group to another agent platform. The group selection is based on the communication dependencies between agent groups and agent platforms. Specifically the communication dependency D_{ij} between local agent group i and agent platform j is decided by summing the communication dependencies between all agents in the local agent group and the agent platform. Let A_i be the set of indexes of all agents in the agent group i .

$$D_{ij} = \sum_{k \in A_i} C_{kj}(t) \quad (4)$$

where $C_{kj}(t)$ is the communication dependency between agent k and agent platform j at time t .

The agent group which has the least dependency to the current agent platform is selected using equation 5.

$$\arg \max_i \left(\frac{\sum_{j, j \neq n} D_{ij}}{D_{in}} \right) \quad (5)$$

where n is the index of the current agent platform.

The destination agent platform of the selected agent group i is decided by the communication dependency between the agent group and agent platforms using equation 6.

$$\arg \max_j (D_{ij}) \quad \text{where } j \neq n \quad (6)$$

Negotiation Phase — If one agent group and its destination agent platform are decided, the Actor Allocation Manager communicates with that of the destination agent platform. If the destination agent platform accepts all agents in the group, the Actor Allocation Manager of the sender agent platform starts the migration phase. Otherwise, this Actor Allocation Manager communicates with that of the second best destination platform until it finds an available destination agent platform or checks the feasibility of all other agent platforms.

This phase of our load sharing mechanism is similar to that of the communication localization mechanism. However, the granularity of negotiation for these two mechanisms is different: the communication localization mechanism is at the level of an agent while the load sharing mechanism is at the level of an agent group. If the destination agent platform has enough space and available computation resource for all agents in the selected local agent group, the Actor Allocation Manager of the destination agent platform can accept the request for the agent group migration. Otherwise, the destination agent platform refuses the request; the destination agent platform cannot accept part of a local agent group.

Agent Migration Phase — When the sender agent platform receives the acceptance reply from the receiver agent platform, the Actor Allocation Manager of the sender agent platform initiates migration of entire agents in the selected local agent group. The following procedure for this phase in the agent distribution mechanism for load sharing is the same as that in the agent distribution mechanism for communication locality.

4. UAV SIMULATIONS

Our dynamic agent distribution mechanisms have been applied to large-scale UAV (Unmanned Aerial Vehicle) simulations. The purpose of these simulations is to analyze the cooperative behavior of micro UAVs under given situations. Several coordination schemes have been simulated and compared to the performance of a selfish UAV scheme. These UAV simulations are based on the

agent-environment interaction model [2]; all UAVs and targets are implemented as intelligent agents, and the navigation space and radar censors of all UAVs are simulated by environment agents. To remove centralized components in distributed computing, each environment agent on a single computer node is responsible for a certain navigation area. In addition to direct communication of UAVs with their names, UAVs may communicate indirectly with other agents through environment agents without agent names. Environment agents use the *ATSpace* model to provide application agent-oriented brokering services [8]. During simulation time, UAVs and targets move from one divided area to another, and they communicate intensively either directly or indirectly.

Figure 2 depicts the components for our UAV simulations. These simulations consist of two types of agents: task-oriented agents and simulation-oriented agents. Task-oriented agents simulate objects in a real situation. For example, a UAV agent represents a physical micro UAV, while a target represents an injured civilian or soldier to be searched and rescued. For simulations, we also need simulation-oriented agents: *Simulation Control Manager* and *Environment Simulator*. Simulation Control Manager synchronizes local virtual times of components, while Environment Simulator simulates both the navigation space and the local broadcasting and radar sensing behavior of all UAVs.

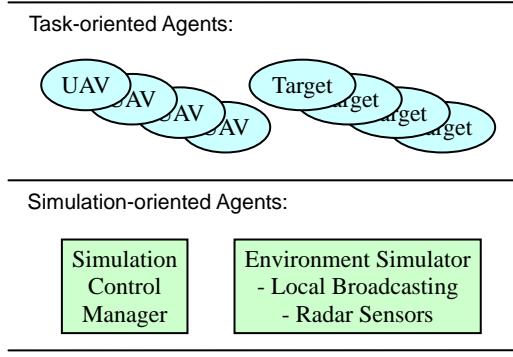


Figure 2 – Architecture of UAV Simulator

When a simulation starts, Simulation Control Manager initializes the parameters of dynamic agent distribution. These parameters include the coefficients for two types of communication dependencies (i.e. agent platform level and agent group level), the migration threshold, the number of local agent groups, and the relative frequency of monitoring and agent grouping phases. During execution, the size of each time step t in dynamic agent distribution is also controlled by Simulation Control Manager; this size is the same as the size of a simulation time step. Thus, the size of time steps varies according to the workload of each simulation step and the processor power. Moreover, the parameters of dynamic agent distribution may be changed by Simulation Control Manager during execution.

5. EXPERIMENTAL RESULTS

We have conducted experiments with micro UAV simulations. These simulations include from 2,000 to 10,000 agents; half of them are UAVs, and the others are targets. Micro UAVs perform a surveillance mission on a mission area to detect and serve moving targets. During the mission time, these UAVs communicate with their neighboring UAVs to perform the mission together. The size of a simulation time step is one half second, and the total simulation time is around 37 minutes. The runtime of each simulation depends on the number of agents and the selected agent distribution mechanism. For these experiments, we use four computers (3.4 GHz Intel CPU and 2 GB main memory) connected by a Giga-bit switch.

Figure 3 and Figure 4 depict the performance benefit of dynamic agent distribution in our experiments comparing with static agent distribution. Even though the dynamic agent distribution mechanisms in our simulations include the overhead from monitoring and decision making, the overall performance of simulations with dynamic agent distribution is much better than that with static agent

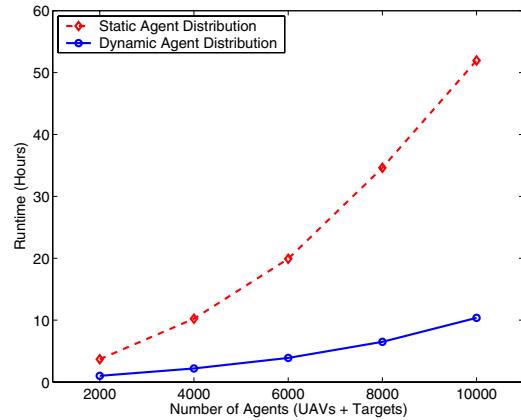


Figure 3 - Runtime of Simulations using Static and Dynamic Agent Distribution

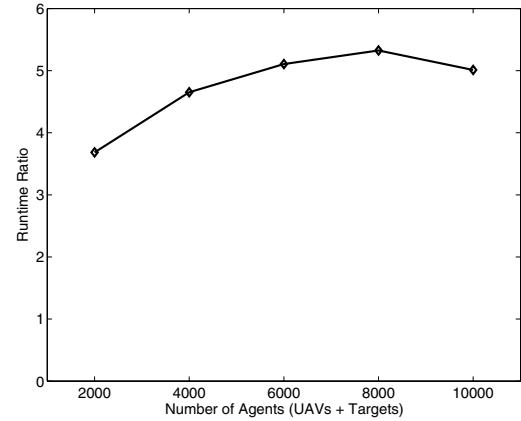


Figure 4 - Runtime Ratio of Static-to-Dynamic Agent Distribution

distribution. In our particular example, as the number of agents is increased, the ratio also generally increases. The simulations using dynamic agent distribution is more than five times faster than those using static agent distribution when the number of agents is large.

6. CONCLUSION

This paper has explained two scalable agent distribution mechanisms used for UAV simulations; these mechanisms distribute agents according to their communication localities and the workload of computer nodes. The main contributions of this research are that our agent distribution mechanisms are based on the dynamic changes of communication localities of agents, that these mechanisms focus on the communication dependencies between agents and agent platforms and the dependencies between agent groups and agent platforms, instead of the communication dependencies among individual agents, and that our mechanisms continuously interact with agent applications to adapt dynamic behaviors of an agent application. In addition, these agent distribution mechanisms are fully distributed mechanisms, are transparent to agent applications, and are concurrently executed with them.

The proposed mechanisms introduce an additional overhead for monitoring and decision making for agent distribution. However, our experiments suggest that this overhead are more than compensated when multi-agent applications have the following attributes: first, an application includes a large number of agents so that the performance on a single computer node is not acceptable; second, some agents communicate more intensively with each other than with other agents, and thus the communication locality of each agent is an important factor in the overall performance of the application; third, the communication patterns of agents are continuously changing, and hence, static agent distribution mechanisms are not sufficient.

In our multi-agent systems, the UAV simulator modifies the parameters of dynamic agent distribution to improve its quality. However, some parameters are currently given by application developers, and finding these values requires developers' skill. We plan to develop learning algorithms to automatically adjust these values during execution of applications.

ACKNOWLEDGEMENTS

The authors would like to thank Sandeep Uttamchandani for his helpful comments and suggestions. This research is sponsored by the Defense Advanced Research Projects Agency under contract number F30602-00-2-0586.

REFERENCES

- [1] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali, "A Load Balancing Framework for Adaptive and Asynchronous Applications," *IEEE Transactions on Parallel and Distributed Systems*, 15(2):183-192, February 2004.
- [2] M. Bouzid, V. Chevrier, S. Vialle, and F. Charpillet, "Parallel Simulation of a Stochastic Agent/Environment Interaction," *Integrated Computer-Aided Engineering*, 8(3):189-203, 2001.
- [3] R.K. Brunner and L.V. Kalé, "Adaptive to Load on Workstation Clusters," *The Seventh Symposium on the Frontiers of Massively Parallel Computations*, pages 106-112, February 1999.
- [4] K. Chow and Y. Kwok, "On Load Balancing for Distributed Multiagent Computing," *IEEE Transactions on Parallel and Distributed Systems*, 13(8):787-801, August 2002.
- [5] T. Desell, K. El Maghraoui, and C. Varela, "Load Balancing of Autonomous Actors over Dynamics Networks," *Hawaii International Conference on System Sciences HICSS-37 Software Technology Track*, Hawaii, January 2004.
- [6] K. Devine, B. Hendrickson, E. Boman, M.St. John, C. Vaughan, "Design of Dynamic Load-Balancing Tools for Parallel Applications," *Proceedings of the International Conference on Supercom0uting*, pages 110-118, Santa Fe, 2001
- [7] L. Gasser and K. Kakugawa, "MACE3J: Fast Flexible Distributed Simulation of Large, Large-Grain Multi-Agent Systems," *Proceedings of the First International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, pages 745-752, Bologna, Italy, July 2002.
- [8] M. Jang, A. Abdel Momen, and G. Agha, "ATSpace: A Middle Agent to Support Application-Oriented Matchmaking and Brokering Services," *IEEE/WIC/ACM IAT(Intelligent Agent Technology)-2004*, pages 393-396, Beijing, China, September 2004.
- [9] M. Jang and G. Agha, Dynamic Agent Allocation for Large-Scale Multi-Agent Applications, *Proceedings of International Workshop on Massively Multi-Agent Systems*, Kyoto, Japan, December 2004.
- [10] K. Popov, V. Vlassov, M. Rafea, F. Holmgren, P. Brand, and S. Haridi, "Parallel Agent-based Simulation on Cluster of Workstations," *Parallel Processing Letters*, 13(4):629-641, 2003.
- [11] D. Sturman, *Modular Specification of Interaction Policies in Distributed Computing*, PhD thesis, University of Illinois at Urbana-Champaign, May 1996.
- [12] C. Walshaw, M. Cross, and M. Everett, "Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes," *Journal of Parallel and Distributed Computing*, 47:102-108, 1997.

Appendix B:

Efficient Agent Communication in Multi-agent Systems

Myeong-Wuk Jang, Amr Ahmed, and Gul Agha

Department of Computer Science
University of Illinois at Urbana-Champaign,
Urbana IL 61801, USA
`{mjang, amrmomen, agha}@uiuc.edu`

Abstract. In open multi-agent systems, agents are mobile and may leave or enter the system. This dynamicity results in two closely related agent communication problems, namely, efficient message passing and service agent discovery. This paper describes how these problems are addressed in the *Actor Architecture (AA)*. Agents in AA obey the operational semantics of actors, and the architecture is designed to support large-scale open multi-agent systems. Efficient message passing is facilitated by the use of dynamic names: a part of the mobile agent name is a function of the platform that currently hosts the agent. To facilitate service agent discovery, middle agents support application agent-oriented matchmaking and brokering services. The middle agents may accept search objects to enable customization of searches; this reduces communication overhead in discovering service agents when the matching criteria are complex. The use of mobile search objects creates a security threat, as codes developed by different groups may be moved to the same middle agent. This threat is mitigated by restricting which operations a migrated object is allowed to perform. We describe an empirical evaluation of these ideas using a large scale multi-agent UAV (Unmanned Aerial Vehicle) simulation that was developed using AA.

1 Introduction

In open agent systems, new agents may be created and agents may move from one computer node to another. With the growth of computational power and network bandwidth, large-scale open agent systems are a promising technology to support coordinated computing. For example, agent mobility can facilitate efficient collaboration with agents on a particular node. A number of multi-agent systems, such as EMAF [3], JADE [4], InfoSleuth [16], and OAA [8], support open agent systems. However, before the vision of scalable open agent systems can be realized, two closely related problems must be addressed:

- *Message Passing Problem*: In mobile agent systems, efficiently sending messages to an agent is not simple because they move continuously from one agent platform to another. For example, the *original agent platform* on which an agent is created should manage the location information about the agent.

However, doing so not only increases the message passing overhead, but it slows down the agent's migration: before migrating, the agent's current host platform must inform the the original platform of the move and may wait for an acknowledgement before enabling the agent.

- *Service Agent Discovery Problem:* In an open agent system, the mail addresses or names of all agents are not globally known. Thus an agent may not have the addresses of other agents with whom it needs to communicate. To address this difficulty, middle agent services, such as *brokering* and *matchmaking* services [25], need to be supported. However, current middle agent systems suffer from two problems: *lack of expressiveness*—not all search queries can be expressed using the middle agent supported primitives; and *incomplete information*—a middle agent does not possess the necessary information to answer a user query.

We address the message passing problem for mobile agents in part by providing a richer name structure: the names of agents include information about their current location. When an agent moves, the location information in its name is updated by the platform that currently hosts the agent. When the new name is transmitted, the location information is used by other platforms to find the current location of that agent if it is the receiver of a message. We address the service agent discovery problem in large-scale open agent systems by allowing client agents to send search objects to be executed in the middle agent address space. By allowing agents to send their own search algorithms, this mitigates both the lack of expressiveness and incomplete information.

We have implemented these ideas in a Java-based agent system called the *Actor Architecture* (or AA). AA supports the *actor semantics* for agents: each agent is an autonomous object with a unique name (address), message passing between agents is asynchronous, new agents may be dynamically created, and agent names may be communicated [1]. AA has been designed with a modular, extensible, and application-independent structure. While AA is being used to develop tools to facilitate large-scale simulations, it may also be used for other large-scale open agent applications. The primary features of AA are: a light-weight implementation of agents, reduced communication overhead between agents, and improved expressiveness of middle agents.

This paper is organized as follows. Section 2 introduces the overall structure and functions of AA as well as the agent life cycle model in AA. Section 3 explains our solutions to reduce the message passing overhead for mobile agents in AA, while Section 4 shows how the search object of AA extends the basic middle agent model. Section 5 desrcies the experimental setting and presents an evaluation of our approaches. Related work is explained in Section 6, and finally, Section 7 concludes this paper with future research directions.

2 The Actor Architecture

AA provides a light-weight implementation of agents as active objects or actors [1]. Agents in AA are implemented as threads instead of processes. They

use object-based messages instead of string-based messages, and hence, they do not need to parse or interpret a given string message, and may use the type information of each field in a delivered message. The actor model provides the infrastructure for a variety of agent systems; actors are social and reactive, but they are *not* explicitly required to be “autonomous” in the sense of being proactive [28]. However, autonomous actors may be implemented in AA, and many of our experimental studies require proactive actors. Although the term agent has been used to mean proactive actors, for our purposes the distinction is not critical. In this paper, we use the terms ‘agent’ and ‘actor’ as synonyms.

The Actor Architecture consists of two main components:

- *AA platforms* which provide the system environment in which actors exist and interact with other actors. In order to execute actors, each computer node must have one AA platform. AA platforms provide actor state management, actor communication, actor migration, and middle agent services.
- *Actor library* which is a set of APIs that facilitate the development of agents on the AA platforms by providing the user with a high level abstraction of service primitives. At execution time, the actor library works as the interface between actors and their respective AA platforms.

An AA platform consists of eight components (see Fig. 1): Message Manager, Transport Manager, Transport Sender, Transport Receiver, Delayed Message Manager, Actor Manager, Actor Migration Manager, and ATSpace.

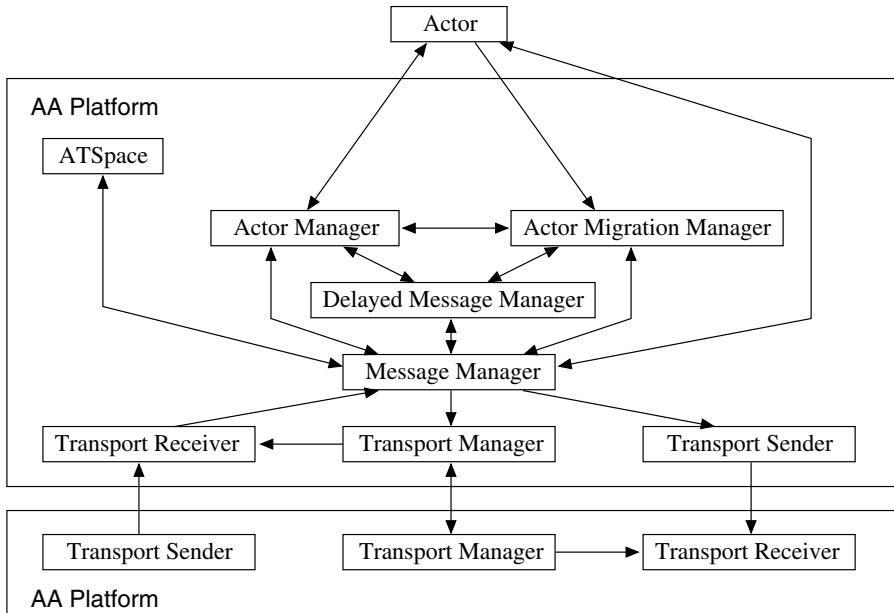


Fig. 1. Architecture of an AA Platform.

The *Message Manager* handles message passing between actors. Every message passes through at least one Message Manager. If the receiver actor of a message exists on the same AA platform, the Message Manager of that platform directly delivers the message to the receiver actor. However, if the receiver actor is not on the same AA platform, this Message Manager delivers the message to the Message Manager of the platform where the receiver currently resides, and finally that Message Manager delivers the message to the receiver actor. The *Transport Manager* maintains a public port for message passing between different AA platforms. When a sender actor sends a message to another actor on a different AA platform, the *Transport Sender* residing on the same platform as the sender receives the message from the Message Manager of that platform and delivers it to the *Transport Receiver* on the AA platform of the receiver. If there is no built-in connection between these two AA platforms, the Transport Sender contacts the Transport Manager of the AA platform of the receiver actor to open a connection so that the Transport Manager can create a Transport Receiver for the new connection. Finally, the Transport Receiver receives the message and delivers it to the Message Manager on the same platform.

The *Delayed Message Manager* temporarily holds messages for mobile actors while they are moving from one AA platform to another. The *Actor Manager* of an AA platform manages the state of actors that are currently executing as well as the locations of mobile actors created on this platform. The *Actor Migration Manager* manages actor migration.

The *ATSpace* provides middle agent services, such as matchmaking and brokering services. Unlike other system components, an *ATSpace* is implemented as an actor. Therefore, any actor may create an *ATSpace*, and hence, an AA platform may have more than one *ATSpaces*. The *ATSpace* created by an AA platform is called the *default ATSpace* of the platform, and all actors can obtain the names of default *ATSpaces*. Once an actor has the name of an *ATSpace*, the actor may send the *ATSpace* messages in order to use its services for finding other actors that match a given criteria.

In AA, actors are implemented as active objects and are executed as threads; actors on an AA platform are executed with that AA platform as part of one process. Each actor has one actor life cycle state on one AA platform at any time (see Fig. 2). When an actor exists on its original AA platform, its state information appears within only its original AA platform. However, the state of an actor migrated from its original AA platform appears both on its original AA platform and on its current AA platform. When an actor is ready to process a message its state becomes **Active** and stays so while the actor is processing the message. When an actor initiates migration, its state is changed to **Transit**. Once the migration ends and the actor restarts, its state becomes **Active** on the new AA platform and **Remote** on the original AA platform. Following a user request, an actor in the **Active** state may move to the **Suspended** state.

In contrast to other agent life cycle models (e.g. [10, 18]), the AA life cycle model uses the **Remote** state to indicate that an actor that was created on the current AA platform is working on another AA platform.

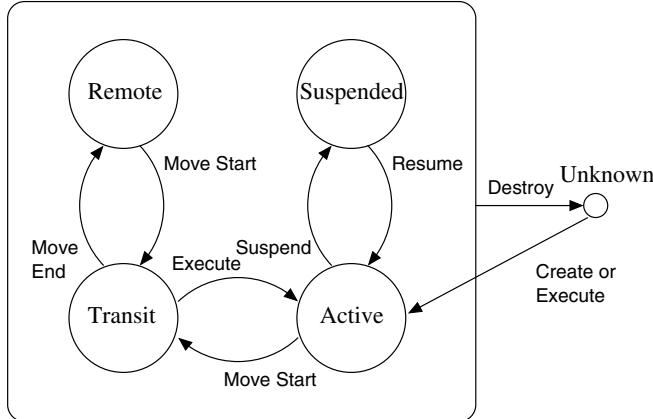


Fig. 2. Actor Life Cycle Model.

3 Optimized Message Delivery

We describe the message delivery mechanisms used to support inter-actor communications. Specifically, AA uses two approaches to reduce the communication overhead for mobile actors that are not on their original AA platforms: *location-based message passing* and *delayed message passing*.

3.1 Location-Based Message Passing

Before an actor can send messages to other actors, it should know the names of the intended receiver actors. In AA, each actor has its own unique name called *UAN* (*Universal Actor Name*). The UAN of an actor includes the *location information* and the *unique identification number* of the actor as follows:

`uan://128.174.245.49:37`

From the above name, we can infer that the actor exists on the host whose IP address is 128.174.245.49, and that the actor is distinguished from other actors on the same platform with its unique identification number 37.

When the *Message Manager* of a sender actor receives a message whose receiver actor has the above name, it checks whether the receiver actor exists on the same AA platform. If they are on the same AA platform, the Message Manager finds the receiver actor on this AA platform and directly delivers the message. Otherwise, the Message Manager of the sender actor delivers the message to the Message Manager of the receiver actor. In order to find the AA platform where the Message Manager of the receiver actor exists, the location information 128.174.245.49 in the UAN of the receiver actor is used. When the Message Manager on the AA platform with IP address 128.174.245.49 receives the message, it finds the receiver actor there and delivers the message.

The above actor naming and message delivery scheme works correctly when all actors are on their original AA platforms. However, because an actor may

migrate from one AA platform to another, we extend the basic behavior of the Message Manager with a *forwarding* service: when a Message Manager receives a message for an actor that has migrated, it delivers the message to the current AA platform of the mobile actor. To facilitate this service, each AA platform maintains the current locations of actors that were created on it, and updates the location information of actors that have come from other AA platforms on their original AA platforms.

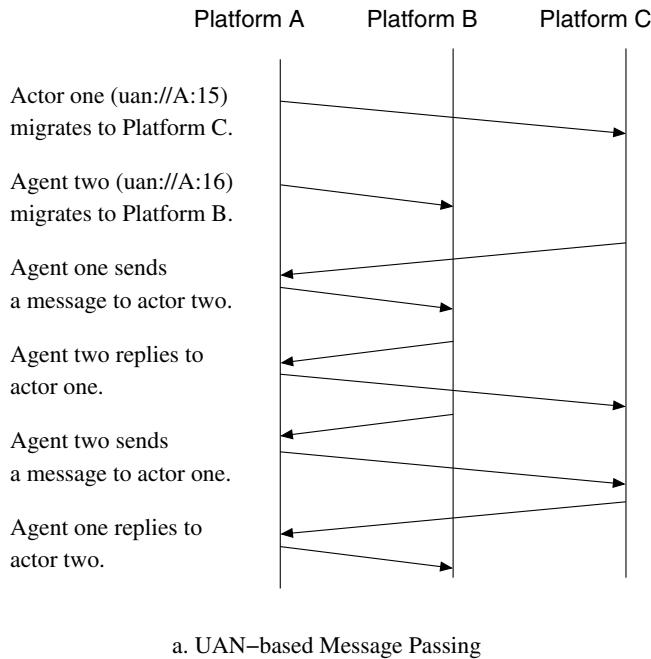
The problem with using only universal actor names for message delivery is that every message for a migrated actor still has to pass through the original AA platform in which the actor was created (Fig. 3.a). This kind of blind indirection may happen even in situations where the receiver actor is currently on an AA platform that is near the AA platform of the sender actor. Since message passing between actor platforms is relatively expensive, AA uses *Location-based Actor Name (LAN)* for mobile actors in order to generally eliminate the need for this kind of indirection. Specifically, the LAN of an actor consists of its current location and its UAN as follows:

```
lan://128.174.244.147//128.174.245.49:37
```

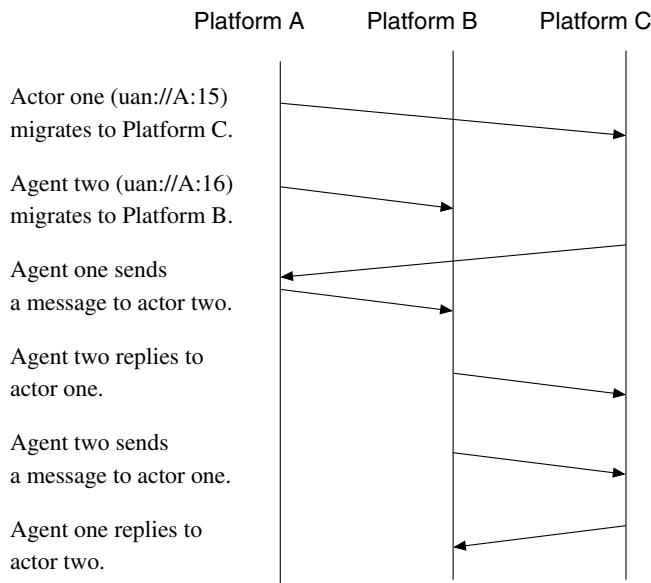
The current location of a mobile actor is set by an AA platform when the actor arrives on the AA platform. If the current location is the same as the location where an actor was created, the LAN of the actor does not have any special information beyond its UAN.

Under the location-based message passing scheme, when the Message Manager of a sender actor receives a message for a remote actor, it extracts the current location of the receiver actor from its LAN and delivers the message to the AA platform where the receiver actor exists. The rest of the procedure for message passing is similar to that in the UAN-based message passing scheme. Fig. 3.b shows how the location-based message passing scheme works. Actor *one* with `ual://C//A:15` sends its first message to actor *two* through the original AA platform of actor *two* because actor *one* does not know the location of actor *two*. This message includes the location information about actor *one* as the sender actor. Therefore, when actor *two* receives the message, it knows the location of actor *one*, and it can now directly send a message to actor *one*. Similarly, when actor *one* receives a message from actor *two*, it learns the location of actor *two*. Finally, the two actors can directly communicate with each other without mediation by their original AA platforms.

In order to use the LAN address scheme, the location information in a LAN should be recent. However, mobile actors may move repeatedly, and a sender actor may have old LANs of mobile actors. Thus a message for a mobile actor may be delivered to its *previous AA platform* from where the actor left. This problem is addressed by having the old AA platform deliver the message to the original AA platform where the actor was created; the original platform always manages the current addresses of its actors. When the receiver actor receives the message delivered through its original AA platform, the actor may send a null



a. UAN-based Message Passing



b. Location-based Message Passing

Fig. 3. Message Passing between Mobile Actors.

message with its LAN to update its location at the sender actor. Therefore, the sender actor can use the updated information for subsequent messages.

3.2 Delayed Message Passing

While a mobile actor is moving from one AA platform to another, the current AA platform of the actor is not well defined. In AA, because the location information of a mobile actor is updated after it finishes migration, its original AA platform thinks the actor still exists on its old AA platform during migration. Therefore, when the Message Manager of the original AA platform receives a message for a mobile actor, it sends the message to the Message Manager of the old AA platform thinking that it is still there. After the Message Manager of the old AA platform receives the message, it forwards the message to the Message Manager of the original AA platform. Thus, a message is continuously passed between these two AA platforms until the mobile actor updates the Actor Manager of its original AA platform with its new location.

In order to avoid unnecessary message thrashing, we use the *Delayed Message Manager* in each AA platform. After the actor starts its migration, the Actor Manager of the old AA platform changes its state to be `Transit`. From this moment, the Delayed Message Manager of this platform holds messages for this mobile actor until the actor reports that its migration has ended. After the mobile actor finishes its migration, its new AA platform sends its old AA platform and its original AA platform a message to inform them that the migration process has ended. When these two AA platforms receive this message, the original AA platform changes the state of the mobile actor from `Transit` to `Remote` while the old AA platform removes all information about the mobile actor, and the Delayed Message Manager of the old AA platform forwards the delayed messages to the Message Manager of the new AA platform of the actor.

4 Active Brokering Service

An ATSpace supports *active brokering services* by allowing agents to send their own search algorithms to be executed in the ATSpace address space [14]. We compare this service to current middle agent services.

Many middle agents are based on *attribute-based communication*. Service agents register themselves with the middle agent by sending a tuple whose attributes describe the service they advertise. To find the desired service agents, a client agent supplies a tuple template with constraints on attributes. The middle agent then tries to find service agents whose registered attributes match the supplied constraints. Systems vary more or less according to the types of constraints (primitives) they support. Typically, a middle agent provides exact matching or regular expression matching [2, 11, 17]. As we mentioned earlier, this solution suffers from a lack of expressiveness and incomplete information.

For example, consider a middle agent with information about seller agents. Each service agent (seller) advertises itself with the following attributes <actor

`name, seller city, product name, product price`. A client agent with the following query is stuck:

*Q1: What are the **best two** (in terms of price) sellers that offer computers and whose locations are roughly **within 50 miles of me**?*

Considering the current tuple space technology, the operator “best two” is clearly not supported (expressiveness problem). Moreover, the tuple space does not include distance information between cities (incomplete information problem). Faced with these difficulties, a user with this complex query Q1 has to transform it into a simpler one that is accepted by the middle agent which retrieves a superset of the data to be retrieved by Q1. In our example, a simpler query could be:

Q2: Find all tuples about sellers that sell computers.

An apparent disadvantage of the above approach is the movement of a large amount of data from the middle agent space to the buyer agent, especially if Q2 is semantically distant from Q1. In order to reduce communication overhead, ATSpace allows a sender agent to send its own search algorithm to find service agents, and the algorithm is executed in the ATSpace. In our example, the buyer agent would send a search object that would inspect tuples in the middle agent and select the best two sellers that satisfy the buyer criteria.

4.1 Security Issues

Although active brokering services mitigate the limitations of middle agents, such as brokers or matchmakers, they also introduce the following security problems in ATSpaces:

- *Data Integrity*: A search object may not modify tuples owned by other actors.
- *Denial of Service*: A search object may not consume too much processing time or space of an ATSpace, and a client actor may not repeatedly send search objects to overload an ATSpace.
- *Illegal Access*: A search object may not carry out unauthorized accesses or illegal operations.

We address the first problem by preventing the search object from modifying tuple data of other actors. This is done by supplying methods of the search object with a copy of the data in the ATSpace. However, when the number of tuples in the ATSpace is large, this solution requires extra memory and computation resources. Thus the ATSpace supports the option of delivering a shallow copy of the original tuples to the search object at the risk of data being changed by search objects as such scheme may compromise the data integrity.

To prevent malicious objects from exhausting the ATSpace computational resource, we deploy user-level thread scheduling as depicted in Fig. 4. When a search object arrives, the object is executed as a thread and its priority is

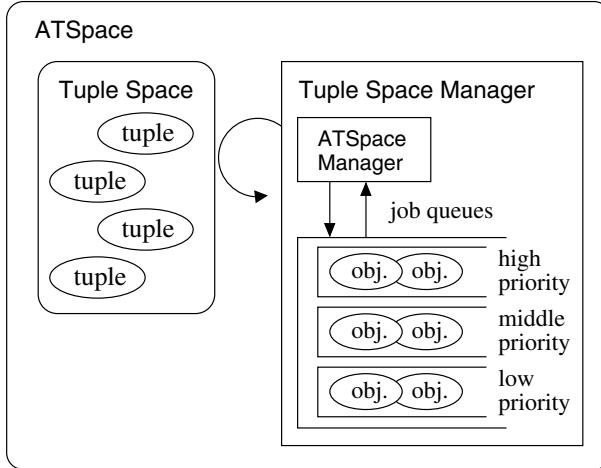


Fig. 4. Architecture of an ATSpace.

set to high. If the thread executes for a long time, its priority is continuously downgraded. Moreover, if the running time of a search object exceeds a certain limit, it may be destroyed by the tuple space manager.

To prevent unauthorized accesses, if the ATSpace is created with an access key, then this key must accompany every message sent from client actors. In this case, actors are allowed only to modify their own tuples. This prevents removal or modification of tuples by unauthorized actors.

5 Experiments and Evaluation

The AA platforms and actors have been implemented in Java language to support operating system independent actor mobility. The Actor Architecture is being used for large-scale UAV (Unmanned Aerial Vehicle) simulations. These simulations investigate the effects of different collaborative behaviors among a large number of micro UAVs during their surveillance missions over a large number of moving targets [15]. For our experiments, we have tested more than 1,000 actors on four computers: 500 micro UAVs, 500 targets, and other simulation purpose actors are executed. The following two sections evaluate our solutions.

5.1 Optimized Message Delivery

According to our experiments, the location-based message passing scheme in AA reduces the number of hops (over AA platforms) that a message for a mobile actor goes through. Since an agent has the location information about its collaborating agents, the agent can carry this information when it moves from one AA platform to another. With location-based message passing, the system is more fault-tolerant; since messages for a mobile actor need not pass through the original AA platform of the actor, the messages may be correctly delivered to the actor even when the actor's original AA platform is not working correctly.

Moreover, delayed message passing removes unnecessary message thrashing for moving agents. When delayed message passing is used, the old AA platform of a mobile actor needs to manage its state information until the actor finishes its migration, and the new platform of the mobile actor needs to report the migration state of the actor to its old AA platforms. In our experiments, this overhead is more than compensated; without delayed message passing the same message may get delivered seven or eight times between the original AA platform and the old AA platform while a mobile actor is moving. If a mobile actor takes more time for its migration, this number may be even greater.

5.2 Active Brokering Service

The performance benefit of ATSpace can be measured by comparing its active brokering services with the data retrieval services of a template-based general middle agent supporting the same service along four different dimensions: the number of messages, the total size of messages, the total size of memory space on the client and middle agent AA platforms, and the computation time for the whole operation. To analytically evaluate ATSpaces, we will use the scenario mentioned in section 4 where a service requesting agent has a complex query that is not supported by the template-based model.

First, with the template-based service, the number of messages is $n + 2$ where n is the number of service agents that satisfy a complex query. This is because the service requesting agent has to first send a message to the middle agent to bring a superset of its final result. This costs two messages: a service request message to the middle agent (**Service_Request_{template}**) that contains Q2 and a reply message that contains agent information satisfying Q2 (**Service_Reply_{template}**). Finally, the service requesting agent sends n messages to the service agents that match its original criteria. With the active brokering service, the total number of messages is $n + 1$. This is because the service requesting agent need not worry about the complexity of his query and only sends a service request message (**Service_Request_{ATSpace}**) to the ATSpace. This message contains the code that represents its criteria along with the message that should be sent to the agents which satisfy these criteria. The last n messages have the same explanation as in the template-based service.

While the number of messages in the two approaches does not differ that much, the total size of these messages may have a huge difference. In both approaches, a set of n messages needs to be sent to the agents that satisfy the final matching criteria. Therefore, the question of whether or not active brokering services result in bandwidth saving depends on the relative size of the other messages. Specifically the difference in bandwidth consumption (*DBC*) between the template-based middle agent and the ATSpace is given by the following equation:

$$\begin{aligned} DBC = & [size(\text{Service_Request}_{\text{template}}) - \\ & size(\text{Service_Request}_{\text{ATSpace}})] + \\ & size(\text{Service_Reply}_{\text{template}}) \end{aligned}$$

In general, since the service request message in active brokering services is larger as it has the search object, the first component is negative. Therefore, active brokering services will only result in a bandwidth saving if the increase in the size of its service request message is smaller than the size of the service reply message in the template-based service. This is likely to be true if the original query (Q1) is complex such that turning it into a simpler one (Q2) to retrieve a superset of the result would incur a great semantic loss and as such would retrieve much extra agent information from the middle agent.

Third, the two approaches put a conflicting requirement on the amount of space needed on both the client and middle agent machines. In the template-based approach the client agent needs to provide extra space to store the tuples returned by Q2. On the hand, the ATSpace needs to provide extra space to store copies of tuples given to search objects. However, a compromise can be made here as the creator of the ATSpace can choose to use the shallow copy of tuples.

Fourth, the difference in computation times of the whole operation in the two approaches depends on two factors: the time for sending messages and the time for evaluating queries on tuples. The tuples in the ATSpace are only inspected once by the search object sent by the service requesting agent. However, in the template-based middle agent, some tuples are inspected twice. First, in order to evaluate Q2, the middle agent needs to inspect all the tuples that it has. Second, these tuples that satisfy Q2 are sent back to the service requesting agent to inspect them again and retain only those tuples that satisfy Q1. If Q1 is complex then Q2 will be semantically distant from Q1, which in turns has two ramifications. First, the time to evaluate Q2 against all the tuples in the middle agent is small relative to the time needed to evaluate the search object over them. Second, most of the tuples on the middle agent would pass Q2 and be sent back to be re-evaluated by the service requesting agent. This reevaluation has nearly the same complexity as running the search object code. Thus we conclude that when the original query is complex and external communication cost is high, the active brokering service will result in time saving.

Apart from the above analytical evaluation, we have run a series of experiments on the UAV simulation to substantiate our claims. (Interested readers may refer to [13] for more details.) Fig. 5 demonstrates the saving in computational time of an ATSpace compared to a template-based middle agent that provides data retrieval services with the same semantic. Fig. 6 shows the wall clock time ratio of a template-based middle agent to an ATSpace. In these experiments, UAVs use either active brokering services or data retrieval services to find their neighboring UAVs. In both cases, the middle agent includes information about locations of UAVs and targets. In case of the active brokering service, UAVs send search objects to an ATSpace while the UAVs using data retrieval service send tuple templates. The simulation time for each run is around 35 minutes, and the wall clock time depends on the number of agents. When the number of agents is small, the difference between the two approaches is not significant. However, as the number of agents is increased, the difference becomes large.

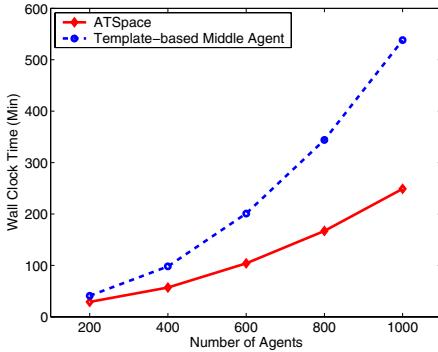


Fig. 5. Wall Clock Time (Min) for ATSpace and Template-based Middle Agent.

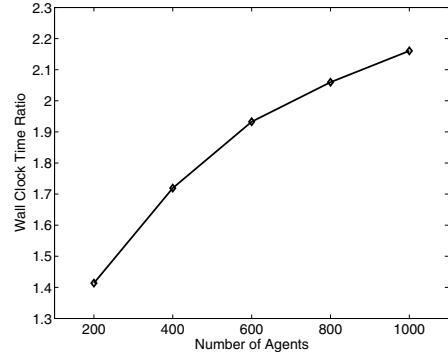


Fig. 6. Wall Clock Time Ratio of Template-based Middle Agent-to-ATSpace.

Fig. 7 depicts the number of messages required in both cases. The number of messages in the two approaches is quite similar but the difference is slightly increased according to the number of agents. Note that the messages increase almost linearly with the number of agents, and that the difference in the number of messages for a template-based middle agent and an ATSpace is small; it is in fact less than 0.01% in our simulations.

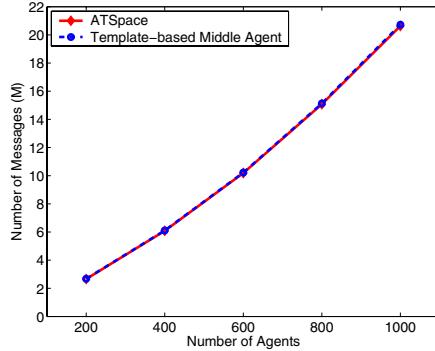


Fig. 7. The Number of Messages for ATSpace and Template-based Middle Agent.

Fig. 8 shows the total message size required in the two approaches, and Fig. 9 shows the total message size ratio. When the search queries are complex, the total message size in the ATSpace approach is much less than that in the template-based middle agent approach. In our UAV simulation, search queries are rather complex and require heavy mathematical calculations, and hence, the ATSpace approach results in a considerable bandwidth saving. It is also interesting to note the relationship between the whole operation time (as shown in Fig. 5) and the bandwidth saving (as shown in Fig. 8). This relationship supports our claim

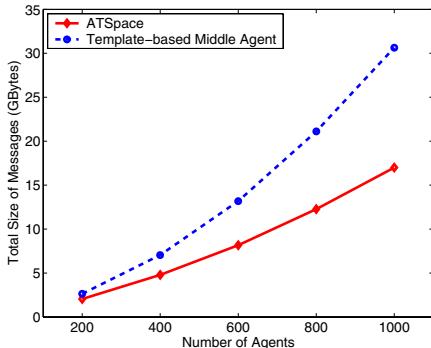


Fig. 8. Total Message Size (GBytes) for ATSpace and Template-based Middle Agent.

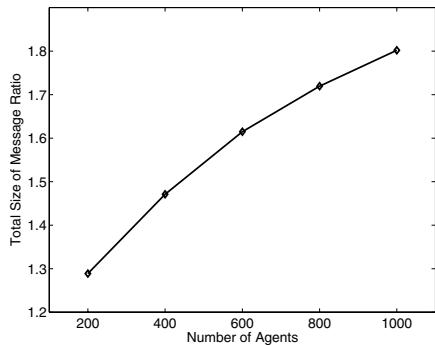


Fig. 9. Total Message Size Ratio for Template-based Middle Agent-to-ATSpace.

that the saving in the total operation time by the ATSpace is largely due to its superiority in efficiently utilizing the bandwidth.

6 Related Work

The basic mechanism of location-based message passing is similar to the message passing in *Mobile IP* [20], although its application domain is different. The original and current AA platforms of a mobile actor correspond to the home and foreign agents of a mobile client in Mobile IP, and the UAN and LAN of a mobile actor are similar to the home address and care-of address of a mobile client in Mobile IP. However, while the sender node in Mobile IP manages a binding cache to map home addresses to care-of addresses, the sender AA platform in AA does not have a mapping table. Another difference is that in mobile IP, the home agent communicates with the sender node to update the binding cache. However, in AA this update can be done by the agent itself when it sends a message that contains its address.

The LAN (Location-based Actor Name) may also be compared to UAL (Universal Actor Locator) in *SALSA* [27]. In SALSA, UAL represents the location of an actor. However, SALSA uses a middle agent called Universal Actor Naming Server to locate the receiver actor. SALSA's approach requires the receiver actor to register its location at a certain middle agent, and the middle agent must manage the mapping table.

The ATSpace approach, which is based on the tuple space model, is related to *Linda* [6]. In the Linda model, processes communicate with other processes through a shared common space called a blackboard or a tuple space without considering references or names of other processes [6, 21]. This approach was used in several agent frameworks, for example *EMAF* [3] and *OAA* [8]. However, these models support only primitive features for pattern-based communication among processes or agents. From the middle agent perspective, *Directory Facilitator* in

the *FIPA* platform [10], *ActorSpace* [2], and *Broker Agent* in *InfoSleuth* [16] are related to our research. However, these systems do not support customizable matching algorithms.

From the expressiveness perspective, some work has been done to extend the matching capability of the basic tuple space model. *Berlinda* [26] allows a concrete entry class to extend the matching function, and *TS* [12] uses policy closures in a Scheme-like language to customize the behavior of tuple spaces. However, these approaches do not allow the matching function to be changed during execution time. At the other hand, *OpenSpaces* [9] provides a mechanism to change matching polices during the execution time. OpenSpaces groups entries in its space into classes and allows each class to have its individual matching algorithm. A manager for each class of entries can change the matching algorithm during execution time. All agents that use entries under a given class are affected by any change to its matching algorithm. This is in contrast to the ATSpace where each agent can supply its own matching algorithm without affecting other agents. Another difference between OpenSpaces and ATSpaces is that the former requires a registration step before putting the new matching algorithm into action, but ATSpace has no such requirement.

Object Space [22] allows distributed applications implemented in the C++ programming language to use a matching function in its template. This matching function is used to check whether an object tuple in the space is matched with the tuple template given in `rd` and `in` operators. However, in the ATSpace the client agent supplied search objects can have a global overview of the tuples stored in the shared space and hence can support global search behavior rather than the one tuple based matching behavior supported in Object Space. For example, using the ATSpace a client agent can find the best ten service agents according to its criteria whereas this behavior cannot be achieved in Object Space.

TuCSoN [19] and *MARS* [5] provide programmable coordination mechanisms for agents through Linda-like tuple spaces to extend the expressive power of tuple spaces. However, they differ in the way they approach the expressiveness problem; while TuCSoN and MARS use reactive tuples to extend the expressive power of tuple spaces, the ATSpace uses search objects to support search algorithms defined by client agents. A reactive tuple handles a certain type of tuples and affects various clients, whereas a search object handles various types of tuples and affects only its creator agent. Therefore, while TuCSoN and MARS extend the general search ability of middle agents, ATSpace supports application agent-oriented searching on middle agents.

Mobile Co-ordination [23] allows agents to move a set of multiple tuple space access primitives to a tuple space for fault tolerance. In *Jada* [7], one primitive may use multiple matching templates. In *ObjectPlaces* [24], dynamic objects are used to change their state whenever corresponding objecplace operations are being called. Although these approaches improve the searching ability of tuple spaces with a set of search templates or dynamic objects, ATSpace provides more flexibility to application agents with their own search code.

7 Conclusion and Future Work

In this paper we addressed two closely related agent communication issues: efficient message delivery and service agent discovery. Efficient message delivery has been addressed using two techniques. First, the agent naming scheme has been extended to include the location information of mobile agents. Second, messages whose destination agent is moving are postponed by the Delayed Message Manager until the agent finishes its migration. For efficient service agent discovery, we have addressed the ATSpace, Active Tuple Space. By allowing application agents to send their customized search algorithms to the ATSpace, application agents may efficiently find service agents. We have synthesized our solutions to the mobile agent addressing and service agent discovery problems in a multi-agent framework.

The long term goal of our research is to build an environment that allows for experimental study of various issues that pertain to message passing and service agent discovery in open multi-agent systems and provide a principled way of studying possible tensions that arise when trying to simultaneously optimize each service. Other future directions include the followings: for efficient message passing, we plan to investigate various trade-offs in using different message passing schemes for different situations. We also plan to extend the Delayed Message Manager to support mobile agents who are continuously moving between nodes. For service agent discovery, we plan to elaborate on our solutions to the security issues introduced with active brokering services.

Acknowledgements

The authors would like to thank the anonymous reviewers and Naqeeb Abbasi for their helpful comments and suggestions. This research is sponsored by the Defense Advanced Research Projects Agency under contract number F30602-00-2-0586.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
2. G. Agha and C.J. Callsen. ActorSpaces: An Open Distributed Programming Paradigm. In *Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming*, pages 23–32, May 1993.
3. S. Baeg, S. Park, J. Choi, M. Jang, and Y. Lim. Cooperation in Multiagent Systems. In *Intelligent Computer Communications (ICC '95)*, pages 1–12, Cluj-Napoca, Romania, June 1995.
4. F. Bellifemine, A. Poggi, and G. Rimassa. JADE - A FIPA-compliant Agent Framework. In *Proceedings of Practical Application of Intelligent Agents and Multi-Agents (PAAM '99)*, pages 97–108, London, UK, April 1999.
5. G. Cabri, L. Leonardi, and F. Zambonelli. MARS: a Programmable Coordination Architecture for Mobile Agents. *IEEE Computing*, 4(4):26–35, 2000.

6. N. Carreiro and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.
7. P. Ciancarini and D. Rossi. Coordinating Java Agents over the WWW. *World Wide Web*, 1(2):87–99, 1998.
8. P.R. Cohen, A.J. Cheyer, M. Wang, and S. Baeg. An Open Agent Architecture. In *AAAI Spring Symposium*, pages 1–8, March 1994.
9. S. Ducasse, T. Hofmann, and O. Nierstrasz. OpenSpaces: An Object-Oriented Framework for Reconfigurable Coordination Spaces. In A. Porto and G.C. Roman, editors, *Coordination Languages and Models, LNCS 1906*, pages 1–19, Limassol, Cyprus, September 2000.
10. Foundation for Intelligent Physical Agents. *SC00023J: FIPA Agent Management Specification*, December 2002. <http://www.fipa.org/specs/fipa00023/>.
11. N. Jacobs and R. Shea. The Role of Java in InfoSleuth: Agent-based Exploitation of Heterogeneous Information Resources. In *Proceedings of Intranet-96 Java Developers Conference*, April 1996.
12. S. Jagannathan. Customization of First-Class Tuple-Spaces in a Higher-Order Language. In *Proceedings of the Conference on Parallel Architectures and Languages - Vol. 2, LNCS 506*, pages 254–276. Springer-Verlag, 1991.
13. M. Jang, A. Ahmed, and G. Agha. A Flexible Coordination Framework for Application-Oriented Matchmaking and Brokering Services. Technical Report UIUCDCS-R-2004-2430, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004.
14. M. Jang, A. Abdel Momen, and G. Agha. ATSpace: A Middle Agent to Support Application-Oriented Matchmaking and Brokering Services. In *IEEE/WIC/ACM IAT(Intelligent Agent Technology)-2004*, pages 393–396, Beijing, China, September 20-24 2004.
15. M. Jang, S. Reddy, P. Totic, L. Chen, and G. Agha. An Actor-based Simulation for Studying UAV Coordination. In *15th European Simulation Symposium (ESS 2003)*, pages 593–601, Delft, The Netherlands, October 26-29 2003.
16. R.J. Bayardo Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments. *ACM SIGMOD Record*, 26(2):195–206, June 1997.
17. D.L. Martin, H. Oohama, D. Moran, and A. Cheyer. Information Brokering in an Agent Architecture. In *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 467–489, London, April 1997.
18. D.G.A. Mobach, B.J. Overeinder, N.J.E. Wijngaards, and F.M.T. Brazier. Managing Agent Life Cycles in Open Distributed Systems. In *Proceedings of the 2003 ACM symposium on Applied Computing*, pages 61–65, Melbourne, Florida, 2003.
19. A. Omicini and F. Zambonelli. TuCSoN: a Coordination Model for Mobile Information Agents. In *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, Pisa, Italy, June 1998.
20. C.E. Perkins. Mobile IP. *IEEE Communications Magazine*, 35:84–99, May 1997.
21. K. Pfleger and B. Hayes-Roth. An Introduction to Blackboard-Style Systems Organization. Technical Report KSL-98-03, Stanford Knowledge Systems Laboratory, January 1998.
22. A. Polze. Using the Object Space: a Distributed Parallel make. In *Proceedings of the 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 234–239, Lisbon, September 1993.

23. A. Rowstron. Mobile Co-ordination: Providing Fault Tolerance in Tuple Space Based Co-ordination Languages. In *Proceedings of the Third International Conference on Coordination Languages and Models*, pages 196–210, 1999.
24. K. Schelfhout and T. Holvoet. ObjectPlaces: An Environment for Situated Multi-Agent Systems. In *Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3 (AAMAS'04)*, pages 1500–1501, New York City, New York, July 2004.
25. K. Sycara, K. Decker, and M. Williamson. Middle-Agents for the Internet. In *Proceedings of the 15th Joint Conference on Artificial Intelligences (IJCAI-97)*, pages 578–583, 1997.
26. R. Tolksdorf. Berlinda: An Object-oriented Platform for Implementing Coordination Language in Java. In *Proceedings of COORDINATION '97 (Coordination Languages and Models)*, LNCS 1282, pages 430–433. Springer-Verlag, 1997.
27. C.A. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices: OOPSLA 2001 Intriguing Technology Track*, 36(12):20–34, December 2001.
28. M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, Ltd, 2002.

Appendix C:

Online Efficient Predictive Safety Analysis of Multithreaded Programs

Koushik Sen and Grigore Roșu and Gul Agha
Department of Computer Science,
University of Illinois at Urbana-Champaign.
`{ksen,grosu,aghag}@cs.uiuc.edu`

Received: date / Revised version: date

Abstract. We present an automated and configurable technique for runtime safety analysis of multithreaded programs which is able to *predict* safety violations from successful executions. Based on a formal specification of safety properties that is provided by a user, our technique enables us to automatically instrument a given program and create an observer so that the program emits *relevant* state update events to the observer and the observer checks these updates against the safety specification. The events are stamped with *dynamic vector clocks*, enabling the observer to infer a *causal partial order* on the state updates. All event traces that are consistent with this partial order, including the actual execution trace, are then analyzed *online* and *in parallel*. A warning is issued whenever one of these potential trace violates the specification. Our technique is scalable and can provide better coverage than conventional testing but its coverage need not be exhaustive. In fact, one can trade-off scalability and comprehensiveness: a *window* in the state space may be specified allowing the observer to infer some of the *more likely* runs; if the size of the window is 1 then only the actual execution trace is analyzed, as is the case in conventional testing; if the size of the window is ∞ then all the execution traces consistent with the actual execution trace are analyzed.

1 Introduction

In multithreaded systems, threads can execute concurrently communicating with each other through a set of shared variables, yielding an inherent potential for subtle errors due to unexpected interleavings. Both rigorous and light-weight techniques to detect errors in multithreaded systems have been extensively investigated. Rigorous techniques include formal methods, such as

model checking and theorem proving, which by exploring –directly or indirectly– all possible thread interleavings, guarantee that a formal model of the system satisfies its safety requirements. Unfortunately, despite impressive recent advances, the size of systems for which model checking or automatic theorem proving is feasible remains rather limited. As a result, most system builders continue to use light-weight techniques such as testing to identify bugs in their implementations.

There are two problems with software testing. First, testing is generally done in an *ad hoc* manner: the software developer must hand-translate the requirements into specific dynamic checks on the program state. Second, test coverage is often rather limited, covering only some execution paths: if an error is not exposed by a particular test case then that error is not detected. To mitigate the first problem, software often includes dynamic checks on a system’s state in order to identify problems at run-time. To ameliorate the second problem, many techniques increase test coverage by developing test-case generation methods that generate test cases which may reveal potential errors with high probability [6, 15, 26].

Based on experience with related techniques and tools, namely JAVA PATHEXPLORER (JPAX) [12] and its sub-system EAGLE [2], we have proposed in [22, 23] an alternative approach, called *predictive runtime analysis*. The essential idea of this analysis technique is as follows. Suppose that a multithreaded program has a safety error, such as a violation of a temporal property, a deadlock, or a data-race. As in testing, we execute the program on some carefully chosen input (a test case). Suppose that the error is not revealed during a particular execution, i.e., the execution is *successful* with respect to that bug. If one regards the execution of a program as a flat, sequential trace of events or states, as in NASA’s JPAX system [12], University of Pennsylvania’s JAVA-MAC [14], Bell Labs’ PET [11], Nokia’s Third Eye framework [16] inspired by Logic Assurance

system [24], or the commercial analysis systems Temporal Rover and DBRover [7–9], then there is not much left to do to find the error except to run another, hopefully better, test case. However, by observing the execution trace in a smarter way, namely as a causal dependency partial order on state updates, we can predict errors that may potentially occur in other possible runs of the multithreaded program.

Our technique merges testing and formal methods to obtain some of the advantages of both while avoiding the pitfalls of *ad hoc* testing and the complexity of full-blown formal verification. Specifically, we develop a *runtime verification* technique for safety analysis of multithreaded systems that can be tuned to analyze a number of traces that are consistent with an actual execution of the program. Two extreme instances of our technique involve checking all or one of the variant traces:

- If all traces are checked then it becomes equivalent to online model checking of an abstract model of the program, called the *multithreaded computation lattice*, extracted from the actual execution trace of the program, like in POTA [19] or JMPaX [22].
- If only one trace is considered, then our technique becomes equivalent to checking just the actual execution of the multithreaded program, as is done in testing or like in other runtime analysis tools like MaC [14] and PaX [12, 2].

In general, depending on the application, one can configure a window within the state space to be explored which, intuitively speaking, provides a causal *distance* from the observed execution within which all traces are exhaustively verified. We call such a window a *causality cone*. An appealing aspect of our technique is that all these traces can be analyzed *online*, as the events are received from the running program, and *in parallel*. The worst case cost of such an analysis is proportional to both the size of the window and the size of the state space of the monitor.

There are three important interrelated components in our runtime verification technique. Our algorithm synthesizes these components automatically from the safety specification:

Instrumentor. The code instrumentor, based on the safety specification, entirely automatically adds code to emit events when *relevant* state updates occur.

Observer. The observer receives the events from the instrumented program as they are generated, enqueues them and then builds a configurable abstract model of the system, known as a computation lattice, on a layer-by-layer basis.

Monitor. As layers are completed, the monitor checks them against the safety specification and then discards them.

The concepts and notions presented in this paper have been experimented and tested on JMPaX 2.0, a

prototype monitoring system for Java programs that we have built. JMPaX 2.0 extends its predecessor JMPaX in at least four non-trivial novel ways:

- The technical notion of *dynamic vector clock* is introduced, which allows us to properly deal with the dynamic creation and destruction of threads.
- The variables that are shared between threads need not be static: an automatic instrumentation technique has been devised that detects automatically when a variable is shared.
- The notion of *cone heuristic*, or *global state window*, is introduced. The cone heuristic enables us to increase the runtime efficiency by analyzing the most likely states in the computation lattice and tune how comprehensive we wish to be.
- The runtime prediction paradigm used is independent of the safety formalism, in the sense that it allows the user to specify any safety property whose bad prefixes can be expressed as a non-deterministic finite automaton (NFA).

Part of this work was presented at the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04) [23].

2 Monitors for Safety Properties

Safety properties are a very important, if not the most important, class of properties that one should consider in monitoring. This is because once a system violates a safety property, there is no way to continue its execution to satisfy the safety property later. Therefore, a monitor for a safety property can precisely say at runtime when the property has been violated, so that an external recovery action can be taken. From a monitoring perspective, what is needed from a safety formula is a succinct representation of its *bad prefixes*, which are finite sequences of states leading to a violation of the property. Therefore, one can abstract away safety properties by languages over finite words.

Automata are a standard means to succinctly represent languages over finite words. In what follows we define a suitable version of automata, called *monitor*, with the property that it has a “bad” state from which it never gets out:

Definition 1. Let \mathcal{S} be a finite or infinite set, that can be thought of as the set of states of the program to be monitored. Then an \mathcal{S} -monitor or simply a *monitor*, is a tuple $\mathcal{M} = \langle \mathcal{M}, m_0, b, \rho \rangle$, where

- \mathcal{M} is the set of states of the monitor;
- $m_0 \in \mathcal{M}$ is the initial state of the monitor;
- $b \in \mathcal{M}$ is the *final state* of the monitor, also called *bad state*; and

- $\rho: \mathcal{M} \times \mathcal{S} \rightarrow 2^{\mathcal{M}}$ is a non-deterministic transition relation with the property that $\rho(b, \Sigma) = \{b\}$ for any $\Sigma \in \mathcal{S}$.

Sequences in \mathcal{S}^* , where ϵ is the empty one, are called (*execution*) *traces*. A trace π is said to be a *bad prefix* in Mon iff $b \in \rho(\{m_0\}, \pi)$, where $\rho: 2^{\mathcal{M}} \times \mathcal{S}^* \rightarrow 2^{\mathcal{M}}$ is recursively defined as $\rho(M, \epsilon) = M$ and $\rho(M, \pi\Sigma) = \rho(\rho(M, \pi), \Sigma)$, where $\rho: 2^{\mathcal{M}} \times \mathcal{S} \rightarrow 2^{\mathcal{M}}$ is defined as $\rho(\{m\} \cup M, \Sigma) = \rho(m, \Sigma) \cup \rho(M, \Sigma)$ and $\rho(\emptyset, \Sigma) = \emptyset$, for all finite $M \subseteq \mathcal{M}$ and $\Sigma \in \mathcal{S}$.

\mathcal{M} is not required to be finite in the above definition, but $2^{\mathcal{M}}$ represents the set of *finite* subsets of \mathcal{M} . In practical situations it is often the case that the monitor is *not* explicitly provided in a mathematical form as above. For example, a monitor can be a specific type of program whose execution is triggered by receiving events from the monitored program; its state can be given by the values of its local variables, and the bad state is a fixed unique state which once reached cannot be changed by any further events.

There are fortunate situations in which monitors can be *automatically generated* from formal specifications, thus requiring the user to focus on system's formal safety requirements rather than on low level implementation details. In fact, this was the case in all the experiments that we have performed so far. We have so far experimented with requirements expressed either in extended regular expressions (ERE) or various variants of temporal logics, with both future and past time. For example, [20, 21] show coinductive techniques to generate minimal static monitors from EREs and from future time linear temporal logics, respectively, and [13, 2] show how to generate dynamic monitors, i.e., monitors that generate their states on-the-fly, as they receive the events, for the safety segment of temporal logic. Note, however, that there may be situations in which the generation of a monitor may not be feasible, even for simple requirements languages. For example, it is well-known that the equivalent automaton of an ERE may be non-elementary larger in the worst case [25]; therefore, there exist relatively small EREs whose monitors cannot even be stored.

Example 1. Consider a reactive controller that maintains the water level of a reservoir within safe bounds. It consists of a water level reader and a valve controller. The water level reader reads the current level of the water, calculates the quantity of water in the reservoir and stores it in a shared variable w . The valve controller controls the opening of a valve by looking at the current quantity of water in the reservoir. A very simple and naive implementation of this system contains two threads: T1, the valve controller, and T2, the water level reader. The code snippet is given in Fig. 1.

Here w is in some proper units such as mega gallons and v is in percentage. The implementation is poorly synchronized and it relies on ideal thread scheduling.

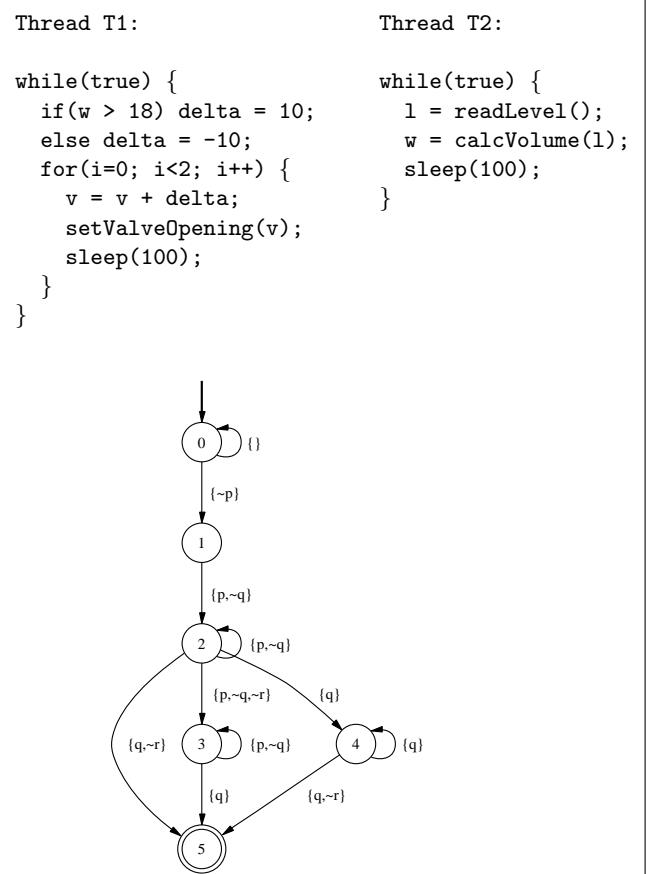


Fig. 1. Two threads (T1 controls the valve and T2 reads the water level) and a monitor.

A sample run of the system can be $\{w = 20, v = 40\}, \{w = 24\}, \{v = 50\}, \{w = 27\}, \{v = 60\}, \{w = 31\}, \{v = 70\}$. As we will see later in the paper, by a run we here mean a sequence of relevant variable writes. Suppose we are interested in a safety property that says “If the water quantity is more than 30 mega gallons, then it is the case that sometime in the past water quantity exceeded 26 mega gallons and since then the valve is open by more than 55% and the water quantity never went down below 26 mega gallon”. We can express this safety property in two different formalisms: linear temporal logic (LTL) with both past-time and future-time operators, or extended regular expressions (EREs) for bad prefixes. The atomic propositions that we will consider are $p : (w > 26), q : (w > 30), r : (v > 55)$. The properties can be written as follows:

$$\begin{aligned}
F_1 &= \square(q \rightarrow ((r \wedge p)\mathcal{S} \uparrow p)) \\
F_2 &= \{\}^* \{\neg p\} \{p, \neg q\}^+ \\
&\quad (\{p, \neg q, \neg r\} \{p, \neg q\}^* \{q\} + \{q\}^* \{q, \neg r\}) \{\}^*
\end{aligned}$$

The formula F_1 in LTL ($\uparrow p$ is a shorthand for “ p and previously not p ”) states that “It is always the case that if $(w > 30)$ then at some time in the past $(w > 26)$ started to be true and since then $(r > 55)$ and $(w > 26)$.”

The formula F_2 characterizes the prefixes that make F_1 false. In F_2 we use $\{p, \neg q\}$ to denote a state where p and $\neg q$ holds and r may or may not hold. Similarly, $\{\}$ represents any state of the system. The monitor automaton for F_2 is given also in Fig. 1.

3 Multithreaded Programs

We consider multithreaded systems in which threads communicate with each other via shared variables. A crucial point is that some variable updates can causally depend on others. We will describe an efficient *dynamic vector clock* algorithm which, given an executing multithreaded program, generates appropriate messages to be sent to an external observer. Section 4 will show how the observer, in order to perform its more elaborated analysis, extracts the state update information from such messages together with the causality partial order.

3.1 Multithreaded Executions and Shared Variables

A multithreaded program consists of n threads t_1, t_2, \dots, t_n that execute concurrently and communicate with each other through a set of shared variables. A *multithreaded execution* is a sequence of events $e_1 e_2 \dots e_r$ generated by the running multithreaded program, each belonging to one of the n threads and having type *internal*, *read* or *write* of a shared variable. We use e_i^j to represent the j^{th} event generated by thread t_i since the start of its execution. When the thread or position of an event is not important we can refer to it generically, such as e , e' , etc.; we may write $e \in t_i$ when event e is generated by thread t_i . Let us fix an arbitrary but fixed multithreaded execution, say \mathcal{C} , and let S be the set of all variables that were shared by more than one thread in the execution. There is an immediate notion of *variable access precedence* for each shared variable $x \in S$: we say e *x-precedes* e' , written $e <_x e'$, iff e and e' are variable access events (reads or writes) to the same variable x , and e “happens before” e' , that is, e occurs before e' in \mathcal{C} . This can be realized in practice by keeping a counter for each shared variable, which is incremented at each variable access.

3.2 Causality and Multithreaded Computations

Let \mathcal{E} be the set of events occurring in \mathcal{C} and let \prec be the partial order on \mathcal{E} :

- $e_i^k \prec e_i^l$ if $k < l$;
- $e \prec e'$ if there is $x \in S$ with $e <_x e'$ and at least one of e, e' is a write;
- $e \prec e''$ if $e \prec e'$ and $e' \prec e''$.

We write $e || e'$ if $e \not\prec e'$ and $e' \not\prec e$. The tuple (\mathcal{E}, \prec) is called the *multithreaded computation* associated with

the original multithreaded execution \mathcal{C} . Synchronization of threads can be easily and elegantly taken into consideration by just generating dummy read/write events when synchronization objects are acquired/released, so the simple notion of multithreaded computation as defined above is as general as practically needed. A permutation of all events e_1, e_2, \dots, e_r that does not violate the multithreaded computation, in the sense that the order of events in the permutation is consistent with \prec , is called a *consistent multithreaded run*, or simply, a *multithreaded run*.

A multithreaded computation can be thought of as the *most general assumption* that an observer of the multithreaded execution can make about the system without knowing what it is supposed to do. Indeed, an external observer simply *cannot disregard* the order in which the same variable is modified and used within the observed execution, because this order can be part of the intrinsic semantics of the multithreaded program. However, multiple consecutive reads of the same variable can be permuted, and the particular order observed in the given execution is not critical. As seen in Section 4, by allowing an observer to analyze *multithreaded computations* rather than just *multithreaded executions*, one gets the benefit of not only properly dealing with potential reorderings of delivered messages (e.g., due to using multiple channels in order to reduce the monitoring overhead), but especially of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling.

3.3 Relevant Causality

Some of the variables in S may be of no importance at all for an external observer. For example, consider an observer whose purpose is to check the property “if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false”; formally, using the interval temporal logic notation in [13], this can be compactly written as $(x > 0) \rightarrow [y = 0, y > z]$. All the other variables in S except x, y and z are essentially irrelevant for this observer. To minimize the number of messages, like in [17] which suggests a similar technique but for distributed systems in which reads and writes are not distinguished, we consider a subset $\mathcal{R} \subseteq \mathcal{E}$ of *relevant events* and define the *\mathcal{R} -relevant causality* on \mathcal{E} as the relation $\triangleleft := \prec \cap (\mathcal{R} \times \mathcal{R})$, so that $e \triangleleft e'$ iff $e, e' \in \mathcal{R}$ and $e \prec e'$. It is important to notice though that the other variables can also indirectly influence the relation \triangleleft , because they can influence the relation \prec . We next provide a technique based on *vector clocks* that correctly implements the relevant causality relation.

3.4 Dynamic Vector Clock Algorithm

We provide a technique based on *vector clocks* [10, 3, 18, 1] that correctly and efficiently implements the relevant

causality relation. Let $V : ThreadId \rightarrow Nat$ be a *partial* map from thread identifiers to natural numbers. We call such a map a *dynamic vector clock (DVC)* because its partiality reflects the intuition that threads are dynamically created and destroyed. To simplify the exposition and the implementation, we assume that each DVC V is a total map, where $V[t] = 0$ whenever V is not defined on thread t .

We associate a DVC with every thread t_i and denote it by V_i . Moreover, we associate two DVCs V_x^a and V_x^w with every shared variable x ; we call the former *access DVC* and the latter *write DVC*. All the DVCs V_i are kept empty at the beginning of the computation, so they do not consume any space. For DVCs V and V' , we say that $V \leq V'$ if and only if $V[j] \leq V'[j]$ for all j , and we say that $V < V'$ iff $V \leq V'$ and there is some j such that $V[j] < V'[j]$; also, $\max\{V, V'\}$ is the DVC with $\max\{V, V'\}[j] = \max\{V[j], V'[j]\}$ for each j . Whenever a thread t_i with current DVC V_i processes event e_i^k , the following algorithm \mathcal{A} is executed:

1. if e_i^k is relevant, i.e., if $e_i^k \in \mathcal{R}$, then

$$V_i[i] \leftarrow V_i[i] + 1$$
2. if e_i^k is a read of a variable x then

$$V_i \leftarrow \max\{V_i, V_x^w\}$$

$$V_x^a \leftarrow \max\{V_x^a, V_i\}$$
3. if e_i^k is a write of a variable x then

$$V_x^w \leftarrow V_x^a \leftarrow V_i \leftarrow \max\{V_x^a, V_i\}$$
4. if e_i^k is relevant then
send message $\langle e_i^k, i, V_i \rangle$ to observer.

In the following discussion we assume a fixed number of threads n . In a program where threads can be created and destroyed dynamically, we only consider the threads that have causally affected the final values of the relevant variables at the end of the computation. For the above algorithm the following result holds:

Lemma 1. *After event e_i^k is processed by thread t_i*

- (a) $V_i[j]$ equals the number of relevant events of t_j that causally precede e_i^k ; if $j = i$ and e_i^k is relevant then this number also includes e_i^k ;
- (b) $V_x^a[j]$ equals the number of relevant events of t_j that causally precede the most recent event in \mathcal{C} that accessed (read or wrote) x ; if $i = j$ and e_i^k is a relevant read or write of x event then this number also includes e_i^k ;
- (c) $V_x^w[j]$ equals the number of relevant events of t_j that causally precede the most recent write event of x ; if $i = j$ and e_i^k is a relevant write of x then this number also includes e_i^k .

To prove the above lemma we introduce some useful formal notation and then state and prove the following two lemmas. For an event e_i^k of thread t_i , let (e_i^k) be the indexed set $\{(e_i^k)_j\}_{1 \leq j \leq n}$, where $(e_i^k)_j$ is the set $\{e_j^l \mid e_j^l \in t_j, e_j^l \in \mathcal{R}, e_j^l \prec e_i^k\}$ when $j \neq i$ and the set $\{e_i^l \mid l \leq k, e_i^l \in \mathcal{R}\}$ when $j = i$. Intuitively, (e_i^k) contains

all the events in the multithreaded computation that causally precede or are equal to e_i^k .

Lemma 2. *With the notation above, for $1 \leq i, j \leq n$:*

1. $(e_j^{l'})_j \subseteq (e_j^l)_j$ if $l' \leq l$;
2. $(e_j^{l'})_j \cup (e_j^l)_j = (e_j^{\max\{l', l\}})_j$ for any l and l' ;
3. $(e_j^l)_j \subseteq (e_i^k)_j$ for any $e_j^l \in (e_i^k)_j$; and
4. $(e_i^k)_j = (e_j^l)_j$ for some appropriate l .

Proof: 1. is immediate, because for any $l' \leq l$, any event e_j^k at thread t_j preceding or equal to $e_j^{l'}$, that is one with $k \leq l'$, also precedes e_j^l .

2. follows by 1., because it is either the case that $l' \leq l$, in which case $(e_j^{l'})_j \subseteq (e_j^l)_j$, or $l \leq l'$, in which case $(e_j^l)_j \subseteq (e_j^{l'})_j$. In either case 2. holds trivially.

3. There are two cases to analyze. If $i = j$ then $e_j^l \in (e_i^k)_j$ if and only if $l \leq k$, so 3. becomes a special instance of 1.. If $i \neq j$ then by the definition of $(e_i^k)_j$ it follows that $e_j^l \prec e_i^k$. Since $e_j^{l'} \prec e_j^l$ for all $l' < l$ and since \prec is transitive, it follows readily that $(e_j^l)_j \subseteq (e_i^k)_j$.

4. Since $(e_i^k)_j$ is a finite set of totally ordered events, it has a maximum element, say e_j^l . Hence, $(e_i^k)_j \subseteq (e_j^l)_j$. By 3., one also has $(e_j^l)_j \subseteq (e_i^k)_j$. \square

Thus, by 4 above, one can uniquely and unambiguously encode a set $(e_i^k)_j$ by just a number, namely the size of the corresponding set $(e_j^l)_j$, i.e., the number of relevant events of thread t_j up to its l^{th} event. This suggests that if the DVC V_i maintained by \mathcal{A} stores that number in its j^{th} component then (a) in Lemma 1 holds.

Before we formally show how reads and writes of shared variables affect the causal dependency relation, we need to introduce some notation. First, since a write of a shared variable introduces a causal dependency between the write event and all the previous read or write events of the same shared variable as well as all the events causally preceding those, we need a compact way to refer at any moment to all the read/write events of a shared variable, as well as the events that causally precede them. Second, since a read event introduces a causal dependency to all the previous write events of the same variable as well as all the events causally preceding those, we need a notation to refer to these events as well. For-

mally, if e_i^k is an event in a multithreaded computation \mathcal{C} and $x \in S$ is a shared variable, then let

$$(e_i^k)_x^a = \begin{cases} \text{The thread-indexed set of all the relevant events that are equal to or causally precede an event } e \text{ accessing } x, \text{ such that } e \text{ occurs before or it is equal to } e_i^k \text{ in } \mathcal{C}, \\ \text{The thread-indexed set of all the relevant events that are equal to or causally precede an event } e \text{ writing } x, \text{ such that } e \text{ occurs before or it is equal to } e_i^k \text{ in } \mathcal{C}. \end{cases}$$

It is obvious that $(e_i^k)_x^w \subseteq (e_i^k)_x^a$. Some or all of the thread-indexed sets of events above may be empty. By convention, if an event, say e , does not exist in \mathcal{C} , then we assume that the indexed sets (e) , $(e)_x^a$, and $(e)_x^w$ are all empty (rather than “undefined”). Note that if V_x^a and V_x^w in \mathcal{A} store the corresponding numbers of elements in the index sets of $(e_i^k)_x^a$ and $(e_i^k)_x^w$ immediately after event e_i^k is processed by thread t_i , respectively, (b) and (c) in Lemma 1 hold.

Even though the sets of events (e_i^k) , $(e_i^k)_x^a$ and $(e_i^k)_x^w$ have mathematically clean definitions, they are based on total knowledge of the multithreaded computation \mathcal{C} . Unfortunately, \mathcal{C} can be very large in practice, so the computation of these sets may be inefficient if not done properly. Since our analysis algorithms are *online*, we would like to calculate these sets *incrementally*, as the observer receives new events from the instrumented program. A key factor in devising efficient update algorithms is to find equivalent *recursive* definitions of these sets, telling us how to calculate a new set of events from similar sets that have been already calculated at previous event updates.

Let $\{e_i^k\}_i^{\mathcal{R}}$ be the indexed set whose j components are empty for all $j \neq i$ and whose i^{th} component is either the one element set $\{e_i^k\}$ when $e_i^k \in \mathcal{R}$ or the empty set otherwise. With the notation introduced, the following important recursive properties hold:

Lemma 3. Let e_i^k be an event in \mathcal{C} and let e_j^l be the event preceding¹ it in \mathcal{C} . If e_i^k is

1. An internal event then
 $(e_i^k] = (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}}$,
 $(e_i^k)_x^a = (e_j^l)_x^a$, for any $x \in S$,
 $(e_i^k)_x^w = (e_j^l)_x^w$, for any $x \in S$;
2. A read of x event then
 $(e_i^k] = (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (e_j^l)_x^w$,
 $(e_i^k)_x^a = (e_i^k) \cup (e_j^l)_x^a$,
 $(e_i^k)_y^a = (e_j^l)_y^a$, for any $y \in S$ with $y \neq x$,
 $(e_i^k)_z^a = (e_j^l)_z^a$, for any $z \in S$;
3. A write of x event then

¹ If e_i^k is the first event then we can assume that e_j^l does not exist in \mathcal{C} , so by convention all the associated sets of events are empty

$$\begin{aligned} (e_i^k] &= (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (e_j^l)_x^a, \\ (e_i^k)_x^a &= (e_i^k], \\ (e_i^k)_x^w &= (e_i^k], \\ (e_i^k)_y^a &= (e_j^l)_y^a, \text{ for any } y \in S \text{ with } y \neq x, \\ (e_i^k)_y^w &= (e_j^l)_y^w, \text{ for any } y \in S \text{ with } y \neq x. \end{aligned}$$

Proof: 1. For the first equality, first recall that $e_i^k \in (e_i^k]$ if and only if e_i^k is relevant. Therefore, it suffices to show that $e \prec e_i^k$ if and only if $e \prec e_i^{k-1}$ for any relevant event $e \in \mathcal{R}$. Since e_i^k is internal, it cannot be in relation \prec_x with any other event for any shared variable $x \in S$, so by the definition of \prec , the only possibilities are that either e is some event $e_i^{k'}$ of thread t_i with $k' < k$, or otherwise there is such an event $e_i^{k'}$ of thread t_i with $k' < k$ such that $e \prec e_i^{k'}$. Hence, it is either the case that e is e_i^{k-1} (so e_i^{k-1} is also relevant) or otherwise $e \prec e_i^{k-1}$. In any of these cases, $e \in (e_i^{k-1}]$. The other two equalities are straightforward consequences of the definitions of $(e_i^k)_x^a$ and $(e_i^k)_x^w$.

2. Like in the proof of 1., $e_i^k \in (e_i^k]$ if and only if $e_i^k \in \mathcal{R}$, so it suffices to show that for any relevant event $e \in \mathcal{R}$, $e \prec e_i^k$ if and only if $e \in (e_i^{k-1}] \cup (e_j^l)_x^a$. Since e_i^k is a read of $x \in S$ event, by the definition of \prec one of the following must hold:

- $e = e_i^{k-1}$. In this case e_i^{k-1} is also relevant, so $e \in (e_i^{k-1}]$;
- $e \prec e_i^{k-1}$. It is obvious in this case that $e \in (e_i^{k-1}]$;
- e is a write of x event and $e \prec_x e_i^k$. In this case $e \in (e_j^l)_x^a$;
- There is some write of x event e' such that $e \prec e'$ and $e' \prec_x e_i^k$. In this case $e \in (e_j^l)_x^a$, too.

Therefore, $e \in (e_i^{k-1}]$ or $e \in (e_j^l)_x^a$.

Let us now prove the second equality. By the definition of $(e_i^k)_x^a$, one has that $e \in (e_i^k)_x^a$ if and only if e is equal to or causally precedes an event accessing $x \in S$ that occurs before or is equal to e_i^k in \mathcal{C} . Since e_i^k is a read of x , the above is equivalent to saying that either it is the case that e is equal to or causally precedes e_i^k , or it is the case that e is equal to or causally precedes an event accessing x that occurs *strictly before* e_i^k in \mathcal{C} . Formally, the above is equivalent to saying that either $e \in (e_i^k]$ or $e \in (e_j^l)_x^a$. If $y, z \in S$ and $y \neq x$ then one can readily see (like in 1. above) that $(e_i^k)_y^a = (e_j^l)_y^a$ and $(e_i^k)_z^a = (e_j^l)_z^a$.

3. It suffices to show that for any relevant event $e \in \mathcal{R}$, $e \prec e_i^k$ if and only if $e \in (e_i^{k-1}] \cup (e_j^l)_x^a$. Since e_i^k is a write of $x \in S$ event, by the definition of \prec one of the following must hold:

- $e = e_i^{k-1}$. In this case $e_i^{k-1} \in \mathcal{R}$, so $e \in (e_i^{k-1}]$;
- $e \prec e_i^{k-1}$. It is obvious in this case that $e \in (e_i^{k-1}]$;
- e is an access of x event (read or write) and $e \prec_x e_i^k$.

In this case $e \in (e_j^l)_x^a$;

- There is some access of x event e' such that $e \prec e'$ and $e' \prec_x e_i^k$. In this case $e \in (e_j^l)_x^a$, too.

Therefore, $e \in (e_i^{k-1}]$ or $e \in (e_j^l]_x^a$.

For the second equality, note that, as for the second equation in 2., one can readily see that $e \in (e_i^k]_x^a$ if and only if $e \in (e_i^k] \cup (e_j^l]_x^a$. But $(e_j^l]_x^a \subseteq (e_i^k]$, so the above is equivalent to $e \in (e_i^k]$. A similar reasoning leads to $(e_i^k]_x^w = (e_i^k]$. The equalities for $y \neq x$ immediate, because e_i^k has no relation to accesses of other shared variables but x . \square

Since each component set of each of the indexed sets in these recurrences has the form $(e_i^k)_i$ for appropriate i and k , and since each $(e_i^k)_i$ can be safely encoded by its size, one can then safely encode each of the above indexed sets by an n -dimensional DVC; these DVCs are precisely V_i for all $1 \leq i \leq n$ and V_x^a and V_x^w for all $x \in S$. Therefore, (a), (b) and (c) of Lemma 1 holds. An interesting observation is that one can regard the problem of recursively calculating (e_i^k) as a dynamic programming problem. As can often be done in dynamic programming problems, one can reuse space and derive the Algorithm \mathcal{A} . The following theorem states that the DVC algorithm correctly implements causality in multithreaded programs.

Theorem 1. *If $\langle e, i, V \rangle$ and $\langle e', i', V' \rangle$ are two messages sent by \mathcal{A} , then $e \triangleleft e'$ if and only if $V[i] \leq V'[i]$ (no typo: the second i is not an i') if and only if $V < V'$.*

Proof: First, note that e and e' are both relevant. The case $i = i'$ is trivial. Suppose $i \neq i'$. Since, by (a) of Lemma 1, $V[i]$ is the number of relevant events that t_i generated before and including e and since $V'[i]$ is the number of relevant events of t_i that causally precede e' , it is clear that $V[i] \leq V'[i]$ if and only if $e \prec e'$. For the second part, if $e \triangleleft e'$ then $V \leq V'$ follows again by (a) of Lemma 1, because any event that causally precedes e also precedes e' . Since there are some indices i and i' such that e was generated by t_i and e' by $t_{i'}$, and since $e' \not\prec e$, by the first part of the theorem it follows that $V'[i'] > V[i']$; therefore, $V < V'$. For the other implication, if $V < V'$ then $V[i] \leq V'[i]$, so the result follows by the first part of the theorem. \square

4 Runtime Model Generation and Predictive Analysis

In this section we consider what happens at the observer's site. The observer receives messages of the form $\langle e, i, V \rangle$. Because of Theorem 1, the observer can infer the causal dependency between the relevant events emitted by the multithreaded system. We show how the observer can be configured to effectively analyze all possible interleavings of events that do not violate the observed causal dependency *online* and *in parallel*. Only one of these interleavings corresponds to the real execution, the others being all potential executions. Hence, the presented technique can *predict* safety violations from successful executions.

4.1 Multithreaded Computation Lattice

Inspired by related definitions in [1], we define the important notions of relevant multithreaded computation and run as follows. A *relevant multithreaded computation*, simply called *multithreaded computation* from now on, is the partial order on events that the observer can infer, which is nothing but the relation \triangleleft . A *relevant multithreaded run*, also simply called *multithreaded run* from now on, is any permutation of the received events which *does not violate* the multithreaded computation. Our major purpose in this paper is to check safety requirements against *all* (relevant) multithreaded runs of a multithreaded system.

We assume that the relevant events are only writes of shared variables that appear in the safety formulae to be monitored, and that these events contain a pair of the name of the corresponding variable and the value which was written to it. We call these variables *relevant variables*. Note that events can change the state of the multithreaded system as seen by the observer; this is formalized next. A *relevant program state*, or simply a *program state*, is a map from relevant variables to concrete values. Any permutation of events generates a sequence of program states in the obvious way, however, not all permutations of events are valid multithreaded runs. A program state is called *consistent* if and only if there is a multithreaded run containing that state in its sequence of generated program states. We next formalize these concepts. For a given computation, let \mathcal{R} be the set of relevant events and \triangleleft be the \mathcal{R} -relevant causality associated with the computation.

Definition 2 (Consistent Run). For a given permutation of events in \mathcal{R} , say $R = e_1 e_2 \dots e_{|\mathcal{R}|}$, we say that R is a *consistent run* if for all pairs of events e and e' , $e \triangleleft e'$ implies that e appears before e' in R .

Let e_i^k be the k^{th} relevant event generated by the thread t_i since the start of its execution. A *cut* C is a subset of \mathcal{R} such that for all $i \in [1, n]$ if e_i^k is present in C then for all $l < k$, e_i^l is also present in C . A cut is denoted by a tuple $(e_1^{k_1}, e_2^{k_2}, \dots, e_n^{k_n})$ where each entry corresponds to the last relevant event for each thread included in C . If a thread i has not seen a relevant event then the corresponding entry is denoted by e_i^0 . A cut C corresponds to a relevant program state that has been reached after all the events in C have been executed. Such a relevant program state is called a *relevant global multithreaded state*, or simply a *relevant global state* or even just *state*, and is denoted by $\Sigma^{k_1 k_2 \dots k_n}$.

Definition 3 (Consistent Cut). A cut is said to be consistent if for all events e and e'

$$(e \in C) \wedge (e' \triangleleft e) \rightarrow (e' \in C)$$

A consistent global state is the one that corresponds to a consistent cut. A relevant event e_i^l is said to be enabled in a consistent global state $\Sigma^{k_1 k_2 \dots k_n}$ if and only if $C \cup \{e_i^l\}$ is a consistent cut, where C is the consistent cut corresponding to the state $\Sigma^{k_1 k_2 \dots k_n}$. The following proposition holds for an enabled event:

Proposition 1. *A relevant event e_i^l is enabled in a consistent global state $\Sigma^{k_1 k_2 \dots k_n}$ if and only if $l = k_i + 1$ and for all relevant events e , if $e \neq e_i^l$ and $e \triangleleft e_i^l$ then $e \in C$, where C is the consistent cut corresponding to the state $\Sigma^{k_1 k_2 \dots k_n}$.*

Proof. Since e_i^l is enabled in the state $\Sigma^{k_1 k_2 \dots k_n}$, $C \cup \{e_i^l\}$ is a cut. This implies that for all events e_i^k , if $k < l$ then $e_i^k \in C \cup \{e_i^l\}$ and hence $e_i^k \in C$. In particular, all the events $e_i^1, e_i^2, \dots, e_i^{l-1}$ are in C . However, e_i^{l-1} is the last relevant event from thread t_i which is included in C . Therefore, $k_i = l - 1$. Since $e_i^l \in C \cup \{e_i^l\}$, $e \triangleleft e_i^l$, and $C \cup \{e_i^l\}$ is a consistent cut, $e \in C \cup \{e_i^l\}$ (by the definition of consistent cut). Since by assumption $e \neq e_i^l$, we have $e \in C$. \square

An immediate consequence of the above proposition is the following corollary:

Corollary 1. *If C is the consistent cut corresponding to the state $\Sigma^{k_1 k_2 \dots k_n}$ and if e_i^l is enabled in $\Sigma^{k_1 k_2 \dots k_n}$ then the state corresponding to the consistent cut $C \cup \{e_i^l\}$ is $\Sigma^{k_1 k_2 \dots k_{i-1} l k_{i+1} \dots k_n}$ or $\Sigma^{k_1 k_2 \dots k_{i-1} (k_i+1) k_{i+1} \dots k_n}$ and we denote it by $\delta(\Sigma^{k_1 k_2 \dots k_n}, e_i^l)$.*

Here the partial function δ maps a consistent state Σ and a relevant event e enabled in that state to a consistent state $\delta(\Sigma, e)$ which is the result of executing e in Σ . Let Σ^{K_0} be the initial global state, $\Sigma^{00\dots 0}$, which is always consistent. The following result holds:

Lemma 4. *If $R = e_1 e_2 \dots e_{|\mathcal{R}|}$ is a consistent multithreaded run then it generates a sequence of global states $\Sigma^{K_0} \Sigma^{K_1} \dots \Sigma^{K_{|\mathcal{R}|}}$ such that for all $r \in [1, |\mathcal{R}|]$, $\Sigma^{K_{r-1}}$ is consistent, e_r is enabled in $\Sigma^{K_{r-1}}$, and $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$.*

Proof. The proof is by induction on r . By definition the initial state Σ^{K_0} is consistent. Moreover, e_1 is enabled in Σ^{K_0} because the cut C corresponding to the state Σ^{K_0} is empty and hence the cut $C \cup \{e_1\} = \{e_1\}$ is consistent. Since Σ^{K_0} is consistent and e_1 is enabled in Σ^{K_0} , $\delta(\Sigma^{K_0}, e_1)$ is defined. Let $\Sigma^{K_1} = \delta(\Sigma^{K_0}, e_1)$.

Let us assume that $\Sigma^{K_{r-1}}$ is consistent, e_r is enabled in $\Sigma^{K_{r-1}}$, and $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$. Therefore, $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$ is also consistent. Let C be the cut corresponding to Σ^{K_r} . To prove that e_{r+1} is enabled in Σ^{K_r} we have to prove that $C \cup e_{r+1}$ is a cut and it is consistent. Let $e_{r+1} = e_i^l$ for some i and l i.e. e_{r+1} is the l^{th} relevant event of thread t_i . For every event e_i^k such that $k < l$, $e_i^k \triangleleft e_i^l$. Therefore, by the definition of consistent run, e_i^k appears before e_i^l for all $0 < k < l$. This implies

that all e_i^k for $0 < k < l$ are included in C . This proves that $C \cup e_i^l$ is a cut. Since C is a cut, for all events e and e' if $e \neq e_i^l$ then $(e \in C \cup \{e_i^l\}) \wedge (e' \triangleleft e) \rightarrow e' \in C \cup \{e_i^l\}$. Otherwise, if $e = e_i^l$ then by the definition of consistent run, if $e' \triangleleft e_i^l$ then e' appears before e_i^l in R . This implies that e' is included in $C \cup \{e_i^l\}$. Therefore, $C \cup \{e_i^l\}$ is consistent which proves that $e_{r+1} = e_i^l$ is enabled in the state Σ^{K_r} . Since, Σ^{K_r} is consistent and e_{r+1} is enabled in Σ^{K_r} , $\delta(\Sigma^{K_r}, e_{r+1})$ is defined. We let $\delta(\Sigma^{K_r}, e_{r+1}) = \Sigma^{K_{r+1}}$. \square

From now on, we identify the sequences of states $\Sigma^{K_0} \Sigma^{K_1} \dots \Sigma^{K_{|\mathcal{R}|}}$ as above with multithreaded runs, and simply call them *runs*. We say that Σ leads-to Σ' , written $\Sigma \rightsquigarrow \Sigma'$, when there is some run in which Σ and Σ' are consecutive states. Let \rightsquigarrow^* be the reflexive transitive closure of the relation \rightsquigarrow . The set of all consistent global states together with the relation \rightsquigarrow^* forms a *lattice* with n mutually orthogonal axes representing each thread. For a state $\Sigma^{k_1 k_2 \dots k_n}$, we call $k_1 + k_2 + \dots + k_n$ its *level*. A *path* in the lattice is a sequence of consistent global states on increasing level, where the level increases by 1 between any two consecutive states in the path. Therefore, a run is just a path starting with $\Sigma^{00\dots 0}$ and ending with $\Sigma^{r_1 r_2 \dots r_n}$, where r_i is the total number of relevant events of thread t_i .

Therefore, a multithreaded computation can be seen as a lattice. This lattice, which is called *computation lattice* and referred to as \mathcal{L} , should be seen as an *abstract model* of the running multithreaded program, containing the relevant information needed in order to analyze the program. Supposing that one is able to *store* the computation lattice of a multithreaded program, which is a non-trivial matter because it can have an exponential number of states in the length of the execution, one can mechanically model-check it against the safety property.

Let $VC(e_i)$ be the DVC associated with the thread t_i when it generated the event e_i . Given a state $\Sigma^{k_1 k_2 \dots k_n}$ we can associate a DVC with the state (denoted by $VC(\Sigma^{k_1 k_2 \dots k_n})$) such that $VC(\Sigma^{k_1 k_2 \dots k_n})[i] = k_i$ i.e. the i^{th} entry of $VC(\Sigma^{k_1 k_2 \dots k_n})$ is equal to the number of relevant events of thread t_i that has causally effected the state. With this definition the following results hold:

Lemma 5. *If a relevant event e from thread t_i is enabled in a state Σ and if $\delta(\Sigma, e) = \Sigma'$ then $\forall j \neq i: VC(\Sigma)[j] = VC(\Sigma')[j]$ and $VC(\Sigma)[i] + 1 = VC(\Sigma')[i]$.*

Proof. This follows directly from the definition of DVC of a state and Corollary 1.

Lemma 6. *If a relevant event e from thread t_i is enabled in a state Σ then $\forall j \neq i: VC(\Sigma)[j] \geq VC(e)[j]$ and $VC(\Sigma)[i] + 1 = VC(e)[i]$.*

Proof. $VC(\Sigma)[i] + 1 = VC(e)[i]$ follows from Lemma 5. Say $k = VC(e)[j]$ for some $j \neq i$. Then by (a) of Lemma 1 we know that the k^{th} relevant event from

thread t_j causally precedes e i.e. $e_j^k \triangleleft e$. Then by proposition 1 $e_j^k \in C$, where C is the cut corresponding to Σ . This implies that $k \leq VC(\Sigma)[j]$ which proves that $\forall j \neq i: VC(\Sigma)[j] \geq VC(e)[j]$.

Lemma 7. *If $R = e_1e_2\dots e_{|\mathcal{R}|}$ is a consistent multithreaded run generating the sequence of global states $\Sigma^{K_0}\Sigma^{K_1}\dots\Sigma^{K_{|\mathcal{R}|}}$, then $VC(\Sigma^{K_i})$ can be recursively defined as follows:*

$$\begin{aligned} VC(\Sigma^{K_0})[j] &= 0 \quad \text{for all } j \in [1, n] \\ VC(\Sigma^{K_r})[j] &= \max(VC(\Sigma^{K_{r-1}})[j], VC(e_r)[j]) \\ &\quad \text{for all } j \in [1, n] \text{ and } 0 < r \leq |\mathcal{R}| \end{aligned}$$

Proof. $\forall j \in [1, n]: VC(\Sigma^{K_0})[j] = 0$ holds by definition. Let e_r be from thread t_i . By Lemma 4 e_r is enabled in $\Sigma^{K_{r-1}}$. Therefore, by Lemma 6, $\forall j \neq i: VC(\Sigma^{K_{r-1}})[j] \geq VC(e_r)[j]$. This implies that $\forall j \neq i: VC(\Sigma^{K_r})[j] = VC(\Sigma^{K_{r-1}})[j] = \max(VC(\Sigma^{K_{r-1}})[j], VC(e_r)[j])$. Otherwise if $j = i$, by Lemma 6, $VC(\Sigma^{K_{r-1}})[j] + 1 = VC(e_r)[j]$. Therefore, $VC(\Sigma^{K_r})[j] = VC(\Sigma^{K_{r-1}})[j] + 1 = VC(e_r)[j] = \max(VC(\Sigma^{K_{r-1}})[j], VC(e_r)[j])$. This proves that $\forall j: VC(\Sigma^{K_r})[j] = \max(VC(\Sigma^{K_{r-1}})[j], VC(e_r)[j])$.

Corollary 2. *If $R = e_1e_2\dots e_{|\mathcal{R}|}$ is a consistent multithreaded run generating the sequence of global states $\Sigma^{K_0}\Sigma^{K_1}\dots\Sigma^{K_{|\mathcal{R}|}}$, then*

$$VC(\Sigma^{K_r})[i] = \max(VC(e_1)[i], VC(e_2)[i], \dots, VC(e_r)[i]) \\ \text{for all } i \in [1, n] \text{ and } 0 < r \leq |\mathcal{R}|$$

Example 2. Figure 2 shows the causal partial order on relevant events extracted by the observer from the multithreaded execution in Example 1, together with the generated computation lattice. The actual execution, $\Sigma^{00}\Sigma^{01}\Sigma^{11}\Sigma^{12}\Sigma^{22}\Sigma^{23}\Sigma^{33}$, is marked with solid edges in the lattice. Besides its DVC, each global state in the lattice stores its values for the relevant variables, w and v . It can be readily seen on Fig. 2 that the LTL property F_1 defined in Example 1 holds on the sample run of the system, and also that it is not in the language of bad prefixes, F_2 . However, F_1 is violated on some other consistent runs, such as $\Sigma^{00}\Sigma^{01}\Sigma^{02}\Sigma^{12}\Sigma^{13}\Sigma^{23}\Sigma^{33}$. On this particular run $\uparrow p$ holds at Σ^{02} ; however, r does not hold at the next state Σ^{12} . This makes the formula F_1 false at the state Σ^{13} . The run can also be symbolically written as $\{\} \{\} \{p\} \{p\} \{p, q\} \{p, q, r\} \{p, q, r\}$. In the automaton in Fig. 1, this corresponds to a possible sequence of states 00123555. Hence, this string is accepted by F_2 as a bad prefix.

Therefore, by carefully analyzing the computation lattice extracted from a successful execution one can infer safety violations in other possible consistent executions. Such violations give informative feedback to users, such as the lack of synchronization in the example above, and may be hard to find by just ordinary testing. In what follows we propose effective techniques

to analyze the computation lattice. A first important observation is that one can generate it *on-the-fly* and analyze it on a level-by-level basis, discarding the previous levels. However, even if one considers only one level, that can still contain an exponential number of states in the length of the current execution. A second important observation is that the states in the computation lattice are not all equiprobable in practice. By allowing a user configurable *window* of most likely states in the lattice centered around the observed execution trace, the presented technique becomes quite scalable, requiring $O(wm)$ space and $O(twm)$ time, where w is the size of the window, m is the size of the bad prefix monitor of the safety property, and t is the size of the monitored execution trace.

4.2 Level By Level Analysis of the Computation Lattice

A naive observer of an execution trace of a multithreaded program would just check the observed execution trace against the monitor for the safety property, say Mon like in Definition 1, and would maintain at each moment a set of states, say $MonStates$ in \mathcal{M} . When a new event arrives, it would create the next state Σ and replace $MonStates$ by $\rho(MonStates, \Sigma)$. If the bad state b will ever be in $MonStates$ then a property violation error would be reported, meaning that the current execution trace led to a bad prefix of the safety property. Here we assume that the events are received in the order in which they are emitted, and also that the monitor works over the global states of the multithreaded programs. This assumption is essential for the observer to deduce the actual execution of the multithreaded program. The knowledge of the actual execution is used by the observer to apply the causal cone heuristics as described later. The assumption is not necessary if we do not want to use causal cone heuristics. The work in [22] describes a technique for the level by level analysis of the computation lattice without the above assumption.

A smart observer, as said before, will analyze not only the observed execution trace, but also all the other consistent runs of the multithreaded system, thus being able to *predict* violations from successful executions. The observer receives the events from the running multithreaded program in real-time and enqueues them in an event queue Q . At the same time, it traverses the computation lattice level by level and checks whether the bad state of the monitor can be hit by any of the runs up to the current level. We next provide the algorithm that the observer uses to construct the lattice level by level from the sequence of events it receives from the running program.

The observer maintains a list of global states ($CurLevel$), that are present in the current level of the lattice. For each event e in the event queue, it tries to construct a new global state from the set of states in the current level and the event e . If the global state is created

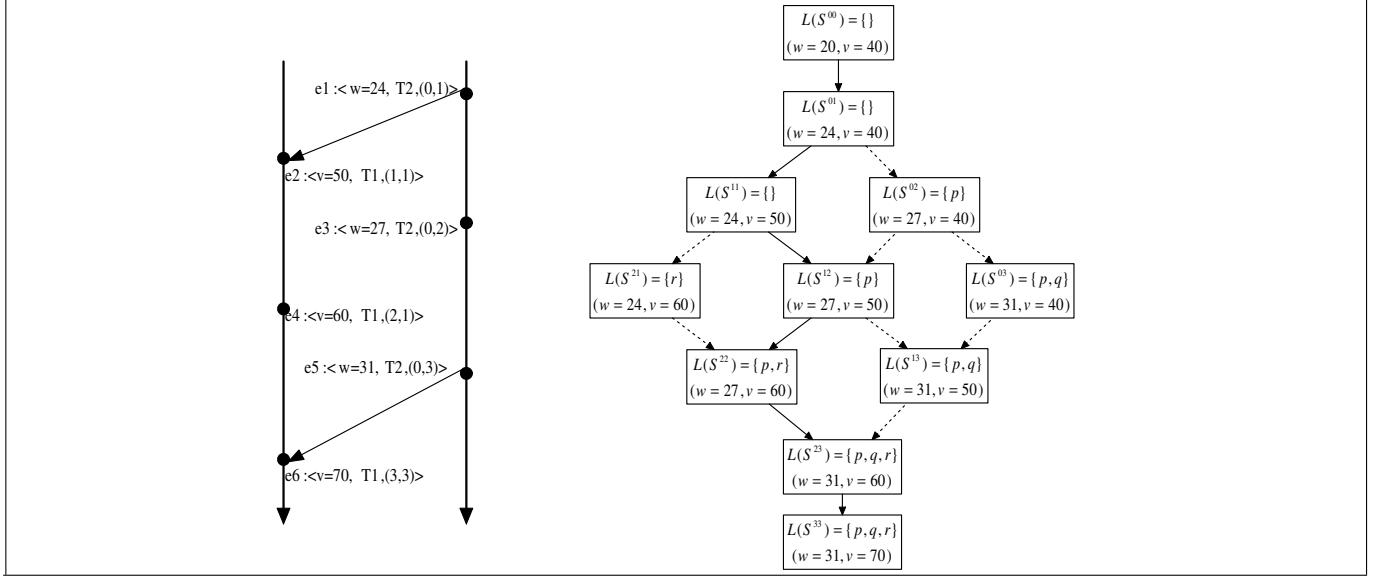


Fig. 2. Computation Lattice

successfully then it is added to the list of global states (*NextLevel*) for the next level of the lattice. The process continues until certain condition, *levelComplete?*() holds. At that time the observer says that the level is complete and starts constructing the next level by setting *CurrLevel* to *NextLevel*, *NextLevel* to empty set, and reallocating the space previously occupied by *CurrLevel*. Here the predicate *levelComplete?*() is crucial for generating only those states in the level that are most likely to occur in other executions, namely those in the *window*, or the *causality cone*, that is described in the next subsection. The *levelComplete?* predicate is also discussed and defined in the next subsection. The pseudo-code for the lattice traversal is given in Fig. 3.

Every global state Σ contains the value of all relevant shared variables in the program, a DVC $VC(\Sigma)$ to represent the latest events from each thread that resulted in that global state. Here the predicate *nextState?*(Σ, e), checks if the event e is enabled in the state Σ , where *threadId*(e) returns the index of the thread that generated the event e , $VC(\Sigma)$ returns the DVC of the global state Σ , and $VC(e)$ returns the DVC of the event e . The correctness of the function is given by Lemma 6. It essentially says that event e can generate a consecutive state for a state Σ , if and only if Σ “knows” everything e knows about the current evolution of the multithreaded system except for the event e itself. Note that e may know less than Σ knows with respect to the evolution of other threads in the system, because Σ has global information.

The function *createState*(Σ, e), which implements the function δ described in Corollary 1 creates a new global state Σ' , where Σ' is a possible consistent global state that can result from Σ after the event e . Together with each state Σ in the lattice, a set of states of the

```

while(not end of computation){
     $Q \leftarrow enqueue(Q, NextEvent())$ 
    while(constructLevel()){}}
}

boolean constructLevel(){
    for each  $e \in Q$  and  $\Sigma \in CurrLevel$  {
        if nextState?( $\Sigma, e$ ) {
             $NextLevel \leftarrow NextLevel \uplus createState(\Sigma, e)$ 
            if levelComplete?( $NextLevel, e, Q$ ) {
                 $Q \leftarrow removeUselessEvents(CurrLevel, Q)$ 
                 $CurrLevel \leftarrow NextLevel$ 
                 $NextLevel \leftarrow \emptyset$ 
                return true}}
        return false
    }
}

boolean nextState?( $\Sigma, e$ ){
     $i \leftarrow threadId(e);$ 
    if ( $\forall j \neq i : VC(\Sigma)[j] \geq VC(e)[j]$  and
         $VC(\Sigma)[i] + 1 = VC(e)[i]$ ) return true
    return false
}

State createState( $\Sigma, e$ ){
     $\Sigma' \leftarrow \text{new copy of } \Sigma$ 
     $j \leftarrow threadId(e); VC(\Sigma')[j] \leftarrow VC(\Sigma)[j] + 1$ 
     $pgmState(\Sigma')[var(e)] \leftarrow value(e)$ 
     $MonStates(\Sigma') \leftarrow \rho(MonStates(\Sigma), \Sigma')$ 
    if  $b \in MonStates(\Sigma')$  {
        output 'property may be violated'
    }
    return  $\Sigma'$ 
}

```

Fig. 3. Level-by-level traversal.

monitor, $\text{MonStates}(\Sigma)$, also needs to be maintained, which keeps all the states of the monitor in which any of the partial runs ending in Σ can lead to. In the function createState , we set the MonStates of Σ' with the set of monitor states to which any of the current states in $\text{MonStates}(\Sigma)$ can transit when the state Σ' is observed. $\text{pgmState}(\Sigma')$ returns the value of all relevant program shared variables in state Σ' , $\text{var}(e)$ returns the name of the relevant variable that is written at the time of event e , $\text{value}(e)$ is the value that is written to $\text{var}(e)$, and $\text{pgmState}(\Sigma')[\text{var}(e) \leftarrow \text{value}(e)]$ means that in $\text{pgmState}(\Sigma')$, $\text{var}(e)$ is updated with $\text{value}(e)$. Lemma 5 justifies that DVC of the state Σ' is updated properly.

The merging operation $\text{nextLevel} \cup \Sigma$ adds the global state Σ to the set nextLevel . If Σ is already present in nextLevel , it updates the existing state's MonStates with the union of the existing state's MonStates and the Monstates of Σ . Two global states are same if their DVCs are equal. Because of the function levelComplete? , it may be often the case that the analysis procedure moves from the current level to the next one before it is exhaustively explored. That means that several events in the queue, which were waiting for other events to arrive in order to generate new states in the current level, become unnecessary so they can be discarded. The function $\text{removeUselessEvents}(\text{CurrLevel}, Q)$ removes from Q all the events that cannot contribute to the construction of any state at the next level. It creates a DVC V_{min} whose each component is the minimum of the corresponding component of the DVCs of all the global states in the set CurrLevel . It then removes all the events in Q whose DVCs are less than or equal to V_{min} . This function makes sure that we do not store any unnecessary events. The correctness of the function is given by the following lemma.

Lemma 8. *For a given relevant event e , if $VC(e) \leq V_{min}$ then $\forall \Sigma \in \text{CurrLevel}$, e is not enabled in Σ .*

Proof. If e is enabled in Σ then by Lemma 6, $VC(e)[i] = VC(\Sigma) + 1$, where t_i is the thread that generated e . This implies that if e is enabled in Σ then $VC(e) \not\leq VC(\Sigma)$. Since $VC(e) \leq V_{min}$ we have $\forall \Sigma \in \text{CurrLevel}$, $VC(e) \leq VC(\Sigma)$. Therefore, e is not enabled in Σ .

The observer runs in a loop till the computation ends. In the loop the observer waits for the next event from the running instrumented program and enqueues it in Q whenever it becomes available. After that the observer runs the function constructLevel in a loop till it returns false. If the function constructLevel returns false then the observer knows that the level is not completed and it needs more events to complete the level. At that point the observer again starts waiting for the next event from the running program and continues with the loop. The pseudo-code for the observer is given at the top of Fig. 3.

4.3 Causality Cone Heuristic

The number of states on a level in the computation lattice can be exponential in the length of the trace. In online analysis, generating all the states in a level may not be feasible. However, note that some states in a level can be considered more likely to occur in a consistent run than others. For example, two independent events that can possibly permute may have a huge time difference. Permuting these two events would give a consistent run, but that run may not be likely to take place in a real execution of the multithreaded program. So we can ignore such a permutation. We formalize this concept as *causality cone*, or *window*, and exploit it in restricting our attention to a small set of states in a given level.

As mentioned earlier we assume that the events are received in an order in which they happen in the computation. We can ensure this linear ordering by executing the DVC algorithm in a synchronized block so that each such execution takes place atomically with respect to each other. Note that this ordering gives the real execution of the program and it respects the partial order associated with the computation. This execution will be taken as a reference in order to compute the most probable consistent runs of the system.

If we consider all the events generated by the executing distributed program as a finite sequence of events, then a lattice formed by any prefix of this sequence is a sub-lattice of the computation lattice \mathcal{L} . This sub-lattice, say \mathcal{L}' , has the following property: if $\Sigma \in \mathcal{L}'$, then for any $\Sigma' \in \mathcal{L}$ if $\Sigma' \rightsquigarrow^* \Sigma$ then $\Sigma' \in \mathcal{L}'$. We can see this sub-lattice as a portion of the computation lattice \mathcal{L} enclosed by a cone. The height of this cone is determined by the length of the current sequence of events. We call this *causality cone*. All the states in \mathcal{L} that are outside this cone cannot be determined from the current sequence of events. Hence, they are outside the causal scope of the current sequence of events. As we get more events this cone moves down by one level.

If we compute a DVC V_{max} whose each component is the maximum of the corresponding component of the DVCs of all the events in the event queue V_{max} represents the DVC of the global state appearing at the tip of the cone. The tip of the cone, by Corollary 2, traverses the actual execution run of the program.

To avoid the generation of a possibly exponential number of states in a given level, we consider a fixed number, say w , of most probable states in a given level. In a level construction, we say that the level is complete once we have generated w states in that level. However, a level may contain less than w states. Then the level construction algorithm gets stuck. Moreover, we cannot determine if a level has less than w states unless we see all the events in the complete computation. This is because we do not know the total number of threads that participate in the computation beforehand. To avoid this scenario we introduce another parameter l , the length of

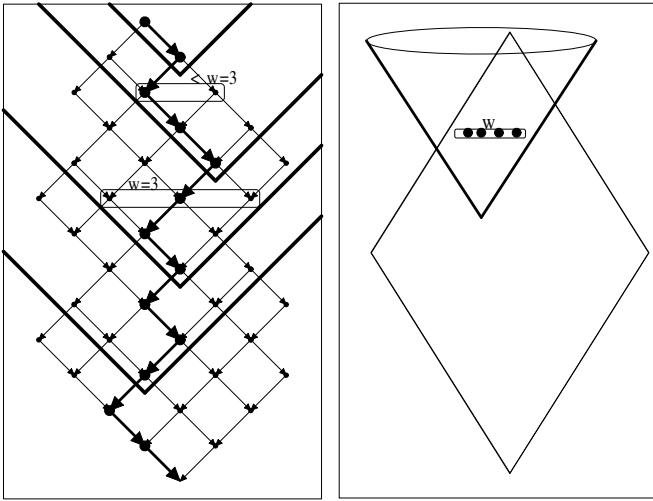


Fig. 4. Causality Cones

```

boolean levelComplete?(NextLevel, e, Q){
    if size(NextLevel) ≥ w then
        return true;
    else if e is the last event in Q
        and size(Q) == l then
        return true;
    else return false;
}

```

Fig. 5. *levelComplete?* predicate

the current event queue. We say that a level is complete if we have used all the events in the event queue for the construction of the states in the current level and the length of the queue is l and we have not crossed the limit w on the number of states. The pseudo-code for *levelComplete?* is given in Fig. 5

Note, here l corresponds to the number of levels of the sub-lattice that can be constructed from the events in the event queue Q . This is because, by Corollary 1, every event in the queue Q generates a state in the next level from a state in the current level in which it is enabled.

Example 3. Figure 6 shows the portion of the computation lattice constructed from the multithreaded execution in Example 1 when the causality cone heuristics is applied with parameters $w = 2$ and $l = 3$. The possible consistent run $\Sigma^{00}\Sigma^{01}\Sigma^{02}\Sigma^{03}\Sigma^{13}\Sigma^{23}\Sigma^{33}$, shown on the left side of the Figure 6, is pruned out by the heuristics. In this particular run the two independent events $e2$ and $e5$ that are permuted have long time difference in the actual execution. Therefore, we can safely ignore this run among all other possible consistent runs.

5 Implementation

We have implemented these new techniques, in version 2.0 of the tool Java MultiPathExplorer (JMPaX), which

has been designed to monitor multithreaded Java programs. The current implementation is written in Java. The tool has three main modules, the *instrumentation* module, the *observer* module and the *monitor* module.

The instrumentation program, named **instrument**, takes a specification file and a list of class files as command line arguments. An example is

```
java instrument spec A.class B.class C.class
```

where the specification file **spec** contains a list of named formulae written in a suitable logic. The program **instrument** extracts the name of the relevant variables from the specification and instruments the classes, provided in the argument, as follows:

- i) For each variable x of primitive type in each class it adds *access* and *write* DVCs, namely `_access_dvc_x` and `_write_dvc_x`, as new fields in the class.
- ii) It adds code to associate a DVC with every newly created thread;
- iii) For each read and write access of a variable of primitive type in any class, it adds codes to update the DVCs according to the algorithm mentioned in Section 3.4;
- iv) It adds code to call a method **handleEvent** of the *observer* module at every write of a relevant variable.

The instrumentation module uses the BCEL [5] Java library to modify Java class files. Currently, the instrumentation module instruments every variable of primitive type in every class. This implies that, during the execution of an instrumented program, the DVC algorithm is executed for every read and write of variables of primitive type. This degrades the performance of the program considerably. We plan to improve the instrumentation by using *escape analysis* [4], to detect the possible shared variables through static analysis. This will reduce the number of instrumentation points and hence will improve the performance.

The *observer* module, that takes two parameters w and l , generates the lattice level-by-level when the instrumented program is executed. Whenever the **handleEvent** method is invoked, it enqueues the event passed as argument to the method **handleEvent**. Based on the event queue and the current level of the lattice, it generates the next level. In the process, it invokes the **nextStates** method (corresponding to ρ in a *monitor*) of the *monitor* module.

The *monitor* module reads the specification file written either as an LTL formula or as a regular expression and generates the non-deterministic automaton corresponding to the formula or the regular expression. It provides the method **nextStates** as an interface to the *observer* module. The method raises an exception if at any point the set of states returned by **nextStates** contains the “bad” state of the automaton. The system be-

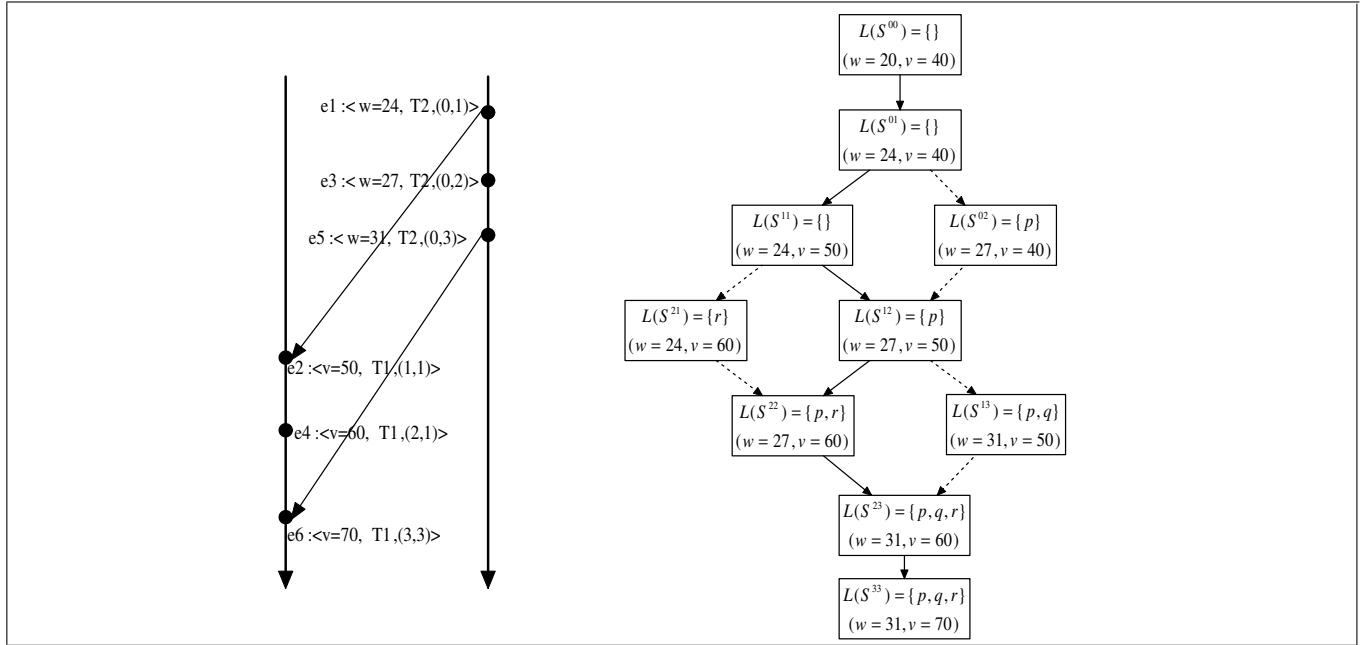


Fig. 6. Causality Cone Heuristics applied to Example 2

ing modular, the user can plug in his/her own *monitor* module for his/her logic of choice.

Since in Java synchronized blocks cannot be interleaved, so corresponding events cannot be permuted, locks are considered as shared variables and a write event is generated whenever a lock is acquired or released. This way, a causal dependency is generated between any exit and any entry of a synchronized block, namely the expected happens-before relation. Java synchronization statements are handled exactly the same way, that is, the shared variable associated to the synchronization object is written at the entrance and at the exit of the synchronized region. Condition synchronizations (wait/notify) can be handled similarly, by generating a write of a dummy shared variable by both the notifying thread before notification and by the notified thread after notification.

Note that the above technique for handing synchronization constructs is conservative as it prevents us from permuting two synchronized blocks even if the events in the two blocks are independent of each other. Future work involves finding an extension of the DVC algorithm that can allow such permutations. This will enable us to extract more interleavings from a computation.

6 Conclusion and Future Work

A formal runtime predictive analysis technique for multithreaded systems has been presented in this paper, in which multiple threads communicating by shared variables are automatically instrumented to send relevant events, stamped by dynamic vector clocks, to an exter-

nal observer which extracts a causal partial order on the global state, updates and thereby builds an abstract runtime model of the running multithreaded system. Analyzing this model on a level-by-level basis, the observer can infer effectively, from a *successful execution* of the observed system, when safety properties can be violated by *other executions*. Attractive future work includes predictions of liveness violations and predictions of data-race and deadlock conditions.

7 Acknowledgments

The work is supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586, the DARPA IXO NEST Program, contract number F33615-01-C-1907), the ONR Grant N00014-02-1-0715, the Motorola Grant MOTOROLA RPS #23 ANT, and the joint NSF/NASA grant CCR-0234524.

References

1. O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.
2. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57, Venice, Italy, January 2004. Springer-Verlag.

3. H. W. Cain and M. H. Lipasti. Verifying sequential consistency using vector clocks. In *Proceedings of the 14th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 153–154. ACM, 2002.
4. J. D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '99)*, volume 34(10) of *SIGPLAN Notices*, pages 1–19, Denver, Colorado, USA, November 1999.
5. M. Dahm. Byte code engineering with the bcel api. Technical Report B-17-98, Freie Universit at Berlin, Institut für Informatik, April 2001.
6. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of Formal Methods Europe (FME'93): Industrial Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284, 1993.
7. D. Drusinsky. Temporal rover. <http://www.time-rover.com>.
8. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
9. D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *Proc. of CAV'03: Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 114–118, Boulder, Colorado, USA, 2003. Springer-Verlag.
10. C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed debugging*, pages 183–194. ACM, 1988.
11. E. L. Gunter, R. P. Kurshan, and D. Peled. PET: An interactive software testing tool. In *Computer Aided Verification (CAV'00)*, volume 1885 of *Lecture Notes in Computer Science*, pages 552–556. Springer-Verlag, 2003.
12. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
13. K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
14. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
15. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
16. R. Lencevicius, A. Ran, and R. Yairi. Third eye - specification-based analysis of software execution traces. In *International Workshop on Automated Program Analysis, Testing and Verification (Workshop of ICSE 2000)*, pages 51–56, June 2000.
17. K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WADG'91)*, volume 579 of *Lecture Notes in Computer Science*, pages 254–272. Springer-Verlag, 1991.
18. F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1989.
19. A. Sen and V. K. .Garg. Partial order trace analyzer (pota) for distrubted programs. In *Proceedings of the 3rd Workshop on Runtime Verification (RV'03)*, Electronic Notes in Theoretical Computer Science, 2003.
20. K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. In *Proceedings of the 3rd Workshop on Runtime Verification (RV'03)*, volume 89 of *ENTCS*, pages 162–181. Elsevier Science, 2003.
21. K. Sen, G. Roşu, and G. Agha. Generating Optimal Linear Temporal Logic Monitors by Coinduction. In *Proceedings of 8th Asian Computing Science Conference (ASIAN'03)*, volume 2896 of *Lecture Notes in Computer Science*, pages 260–275. Springer-Verlag, December 2003.
22. K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proceedings of 4th joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*. ACM, 2003.
23. K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 123–138, Barcelona, Spain, March 2004.
24. O. Shtrichman and R. Goldring. The ‘logic-assurance’ system - a tool for testing and controlling real-time systems. In *Proc. of the Eighth Israeli Conference on computer systems and sofware engineering (ICCSSE97)*, pages 47–55, June 1997.
25. L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). pages 1–9. ACM Press, 1973.
26. S. A. Vilkomir and J. P. Bowen. Formalization of software testing criteria using the Z notation. In *Proceedings of COMPSAC 01: 25th IEEE Annual International Computer Software and Applications Conference*, pages 351–356. IEEE Computer Society, Oct. 2001.

An Instrumentation Technique for Online Analysis of Multithreaded Programs

Grigore Roşu and Koushik Sen
Department of Computer Science,
University of Illinois at Urbana-Champaign, USA
Email: {grosu,ksen}@uiuc.edu

Abstract

Runtime verification of multithreaded systems, that is, the process of finding errors in multithreaded systems as they execute, is the theme of this paper. The major goal of the work in this paper is to present an automatic code instrumentation technique, based on multithreaded vector clocks, for generating the causal partial order on relevant state update events from a running multithreaded program. By means of several examples, it is shown how this technique can be used in a formal testing environment, not only to detect, but especially to predict safety errors in multithreaded programs. The prediction process consists of rigorously analyzing other potential executions that are consistent with the causal partial order: some of these can be erroneous despite the fact that the particular observed execution was successful. The proposed technique has been implemented as part of a Java program analysis tool, called Java MultiPathExplorer and abbreviated JMPAX. A bytecode instrumentation package is used, so the Java source code of the tested programs is not necessary.

1. Introduction and Motivation

A major drawback of testing is its lack of coverage: if an error is not exposed by a particular test case then that error is not detected. To ameliorate this problem, many techniques have been investigated in the literature to increase the coverage of testing, such as test-case generation methods for generating those test cases that can reveal potential errors with high probability [8, 21, 30]. Based on experience with related techniques already implemented in JAVA PATHEXPLORER (JPAX) [15, 14] and its sub-system EAGLE [4], we have proposed in [27, 28] an alternative approach, called “predictive runtime analysis”, which can be intuitively described as follows.

Suppose that a multithreaded program has a subtle safety error, such as a safety or a liveness temporal property violation, or a deadlock or a data-race. Like in testing, one

executes the program on some carefully chosen input (test case) and suppose that, unfortunately, the error is not revealed during that particular execution; such an execution is called *successful* with respect to that bug. If one regards the execution of a program as a flat, sequential trace of events or states, like NASA’s JPAX system [15, 14], University of Pennsylvania’s JAVA-MAC [20], Bell Labs’ PET [13], or the commercial analysis systems Temporal Rover and DBRover [9, 10, 11], then there is not much left to do to find the error except to run another, hopefully better, test case. However, by observing the execution trace in a smarter way, namely as a causal dependency partial order on state updates, one can predict errors that can potentially occur in other possible runs of the multithreaded program.

The present work is an advance in *runtime verification* [16], a scalable complementary approach to the traditional formal verification methods (such as theorem proving and model checking [6]). Our focus here is on multithreaded systems with shared variables. More precisely, we present a simple and effective algorithm that enables an external observer of an executing multithreaded program to detect and predict specification violations. The idea is to properly *instrument* the system before its execution, so that it will emit relevant events at runtime. No particular specification formalism is adopted in this paper, but examples are given using a temporal logic that we are currently considering in JAVA MULTIPATHEXPLORER (JMPAX) [27, 28], a tool for safety violation prediction in Java multithreaded programs which supports the presented technique.

In multithreaded programs, threads communicate via a set of shared variables. Some variable updates can causally depend on others. For example, if a thread writes a shared variable x and then another thread writes y due to a statement $y = x + 1$, then the update of y *causally depends* upon the update of x . Only read-write, write-read and write-write causalities are considered, because multiple consecutive reads of the same variable can be permuted without changing the actual computation. A state is a map assigning values to variables, and a specification consists of properties on these states. Some variables may be of no importance at

all for an external observer. For example, consider an observer which monitors the property “if $(x > 0)$ then $(y = 0)$ ” has been true in the past, and since then $(y > z)$ was always false”. All the other variables except x , y and z are irrelevant for this observer (but they can clearly affect the causal partial ordering). To minimize the number of messages sent to the observer, we consider a subset of *relevant events* and the associated *relevant causality*.

We present an algorithm that, given an executing multithreaded program, generates appropriate messages to be sent to an external observer. The observer, in order to perform its more elaborated system analysis, extracts the state update information from such messages together with the relevant causality partial order among the updates. This partial order abstracts the behavior of the running program and is called *multithreaded computation*. By allowing an observer to analyze multithreaded computations rather than just flat sequences of events, one gets the benefit of not only properly dealing with potential reordering of delivered messages (reporting global state accesses), but also of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling and can be hard, if not impossible, to find by just testing.

To be more precise, let us consider a real-life example where a runtime analysis tool supporting the proposed technique, such as JMPAX, would be able to predict a violation of a property from a single, successful execution of the program. However, like in the case of data-races, the chance of detecting this safety violation by monitoring only the actual run is very low. The example consists of a two threaded program to control the landing of an airplane. It has three variables `landing`, `approved`, and `radio`; their values are 1 when the *plane is landing*, *landing has been approved*, and *radio signal is live*, respectively, and 0 otherwise. The safety property to verify is “If the plane has started landing, then it is the case that landing has been approved and since the approval the radio signal has never been down.”

The code snippet for a naive implementation of this control program is shown in Fig. 1. It uses some dummy functions, `askLandingApproval` and `checkRadio`, which can be implemented properly in a real scenario. The program has a serious problem that cannot be detected easily from a single run. The problem is as follows. Suppose the plane has received approval for landing and just before it started landing the radio signal went off. In this situation, the plane must abort landing because the property was violated. But this situation will very rarely arise in an execution: namely, when `radio` is set to 0 between the approval of landing and the start of actual landing. So a tester or a simple observer will probably never expose this bug. However, note that even if the radio goes off *after* the landing has started, a case which is quite likely to be considered during testing but in which the property is *not* violated, JMPAX

```

int landing = 0, approved = 0, radio = 1;
void thread1(){
    askLandingApproval();
    if(approved==1){
        print("Landing approved");
        landing = 1;
        print("Landing started");}
    else {print("Landing not approved");}
}
void askLandingApproval(){
    if(radio==0) approved = 0
    else approved = 1;

void thread2(){
    while(radio){checkRadio();} }
void checkRadio(){
    possibly change value of radio;
}

```

Figure 1. A buggy implementation of a flight controller.

will still be able to construct a possible run (counterexample) in which `radio` goes off between landing and approval. In Section 4, among other examples, it is shown how JMPAX is able to predict two safety violations from a single successful execution of the program. The user will be given enough information (the entire counterexample execution) to understand the error and to correct it. In fact, this error is an artifact of a bad programming style and cannot be easily fixed - one needs to give a proper event-based implementation. This example shows the power of the proposed runtime verification technique as compared to the existing ones in JPAX and JAVA-MAC.

The main contribution of this paper is a detailed presentation of an instrumentation algorithm which plays a crucial role in extracting the causal partial order from one flat execution, and which is based on an appropriate notion of vector clock inspired from [12, 24], called *multithreaded vector clock (MVC)*. An MVC V is an array associating a natural number $V[i]$ to each thread i in the multithreaded system, which represents the number of relevant events generated by that thread since the beginning of the execution. An MVC V_i is associated to each thread t_i , and two MVCs, V_x^a (access) and V_x^w (write) are associated to each shared variable x . When a thread t_i processes event e , which can be an internal event or a shared variable read/write, the code in Fig. 2 is executed. We prove that \mathcal{A} correctly implements the relevant causal partial order, i.e., that for any two messages $\langle e, i, V \rangle$ and $\langle e', j, V' \rangle$ sent by \mathcal{A} , e and e' are relevant and e causally precedes e' iff $V[i] \leq V'[i]$. This algorithm can be implemented in several ways. In the case of Java, we prefer to implement it as an appropriate instrumentation procedure of code or bytecode, to execute \mathcal{A} whenever a shared variable is accessed. Another implementation could be to modify a JVM. Yet another one would be to en-

ALGORITHM \mathcal{A}

INPUT: event e generated by thread t_i

1. if e is relevant then
 $V_i[i] \leftarrow V_i[i] + 1$
2. if e is a read of a shared variable x then
 $V_i \leftarrow \max\{V_i, V_x^w\}$
 $V_x^a \leftarrow \max\{V_x^a, V_i\}$
3. if e is a write of a shared variable x then
 $V_x^w \leftarrow V_x^a \leftarrow V_i \leftarrow \max\{V_x^a, V_i\}$
4. if e is relevant then
send message $\langle e, i, V_i \rangle$ to observer

Figure 2. The vector clock instrumentation algorithm.

force shared variable updates via library functions, which execute \mathcal{A} as well. All these can add significant delays to the normal execution of programs. The work in this paper has been partly presented at the workshop *Parallel and Distributed Systems: Testing and Debugging (PADTAD'04)* in April 2004. A preliminary version has been published in the PADTAD'04 proceedings [25].

Multithreaded systems are briefly introduced in Section 2 and then the main algorithm is rigorously *derived* from its desired properties in Section 3. Section 4 describes possible uses of the algorithm, and Section 5 concludes the paper.

2. Multithreaded Systems

We consider multithreaded systems in which several threads communicate with each other via a set of shared variables. A crucial point is that some variable updates can causally depend on others. We will present an instrumentation algorithm which, given an executing instrumented multithreaded program, generates appropriate messages to be sent to an external observer. The observer, in order to perform its analysis, extracts the state update information from such messages together with the causality partial order among the updates.

In this paper we only consider a fixed number of threads. However, the presented instrumentation technique can be easily extended to systems consisting of a variable number of threads, where these can be dynamically created and/or destroyed [28].

2.1. Multithreaded Executions

Given n threads t_1, t_2, \dots, t_n , a *multithreaded execution* is a sequence of events $e_1 e_2 \dots e_r$, each belonging to one of the n threads and having type *internal*, *read* or *write* of a shared variable. We use e_i^k to represent the k -th event generated by thread t_i since the start of its execution. When the thread or position of an event is not important, we may refer to it generically, such as e, e' , etc. We may write $e \in t_i$ when event e is generated by thread t_i .

Let us fix an arbitrary but fixed multithreaded execution, say \mathcal{M} , and let S be the set of all shared variables. There is an immediate notion of *variable access precedence* for each shared variable $x \in S$: we say that e *x-precedes* e' , written $e <_x e'$, if and only if e and e' are variable access events (reads or writes) to the same variable x , and e “happens before” e' , that is, e occurs before e' in \mathcal{M} . This “happens-before” relation can be easily realized in practice by keeping a counter for each shared variable, which is incremented at each variable access.

The notion of precedence above is consistent with the *sequential memory model* of multithreaded systems, which is assumed from now on in this paper. In fact, we assume that all shared memory accesses (reads or writes of shared variables in S) are atomic and instantaneous. This will allow us to properly reason about causal dependencies between events. For example, if a thread writes x then writes y followed by another thread writing y then x , under a non-sequential memory model it would be possible that the first write of y takes place before the second, while the first write of x takes place *after* the second write of x , leading to a circular causal dependency between the two threads. So far we have not considered non-sequential memory models in our runtime verification efforts, which admittedly would reveal additional potential errors but would increase the complexity of runtime analysis. This may change in the future if the assumed sequential model will be found too restrictive in practical experiments. Recall that our purpose is to find errors in multithreaded programs, not to prove systems correct. The errors that we detect using the sequential model are also errors in other memory models, so our approach is currently conservative.

2.2. Causality and Multithreaded Computations

Let \mathcal{E} be the set of all the events occurring in the multithreaded execution \mathcal{M} and let \prec be the partial order relation on \mathcal{E} defined as follows:

- $e_i^k \prec e_i^l$ if $k < l$;
- $e \prec e'$ if there is $x \in S$ with $e <_x e'$ and at least one of e, e' is a write;

- $e \prec e''$ if $e \prec e'$ and $e' \prec e''$.

The first item above states that the events of a thread i are causally ordered by their corresponding occurrence time. The second item says that if two events e and e' , of the same thread or not, access a shared variable x and one of them is a write, then the most recent one causally depends on the former one. No causal constraint is imposed on read-read events, so they are permutable. Finally, by closing it under transitivity, the third item defines \prec as the smallest partial order including the first two types of causal constraints. We write $e \parallel e'$ if $e \not\prec e'$ and $e' \not\prec e$. The partial order \prec on \mathcal{E} defined above is called the *multithreaded computation* associated with the original multithreaded execution \mathcal{M} .

As discussed in Section 3.1 in more depth, synchronization of threads can be handled at no additional effort by just generating appropriate read/write events when synchronization objects are acquired/released, so the simple notion of multithreaded computation as defined above is as general as practically needed. A linearization (or a permutation) of all the events e_1, e_2, \dots, e_r that is consistent with the multithreaded computation, in the sense that the order of events in the permutation is consistent with \prec , is called a *consistent multithreaded run*, or simply, a *multithreaded run*. Intuitively, a multithreaded run can be viewed as a possible execution of the same system under a different execution speed of each individual thread.

A multithreaded computation can be thought of as the *most general assumption* that an observer of the multithreaded execution can make about the system without knowing its semantics. Indeed, an external observer simply *cannot disregard* the order in which the same variable is modified and used within the observed execution, because this order can be part of the intrinsic semantics of the multithreaded program. However, multiple causally independent modifications of different variables can be permuted, and the particular order observed in the given execution is not critical. By allowing an observer to analyze *multithreaded computations*, rather than just *multithreaded executions* like JPAX [15, 14], JAVA-MAC [20], and PET [13], one gets the benefit of not only properly dealing with potential reorderings of delivered messages (e.g., due to using multiple channels to reduce the monitoring overhead), but also of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling.

2.3. Relevant Causality

Some variables in S may be of no importance for an external observer. For example, consider an observer whose purpose is to check the property “if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false”; formally, using the interval temporal logic notation notation in [18], this can be compactly written as

$(x > 0) \rightarrow [y = 0, y > z]$. All the other variables in S except x, y and z are essentially irrelevant for this observer. To minimize the number of messages, like in [23] which suggests a similar technique but for distributed systems in which reads and writes are not distinguished, we consider a subset $\mathcal{R} \subseteq \mathcal{E}$ of *relevant events* and define the \mathcal{R} -*relevant causality* on \mathcal{E} as the relation $\triangleleft := \prec \cap (\mathcal{R} \times \mathcal{R})$, so that $e \triangleleft e'$ if and only if $e, e' \in \mathcal{R}$ and $e \prec e'$. It is important to notice though that the other shared variables can also indirectly influence the relation \triangleleft , because they can influence the relation \prec .

We next introduce multithreaded vector clocks (MVC), together with a technique that is proved to correctly implement the relevant causality relation.

3. Multithreaded Vector Clock Algorithm

Inspired and stimulated by the elegance and naturality of vector clocks [12, 24, 2] in implementing causal dependency in distributed systems, we next devise an algorithm to implement the relevant causal dependency relation in multithreaded systems. Since in multithreaded systems communication is realized by shared variables rather than message passing, to avoid any confusion we call the corresponding vector-clock data-structures *multithreaded vector clocks* and abbreviate them (*MVC*). The algorithm presented next has been mathematically derived from its desired properties, after several unsuccessful attempts to design it on a less rigorous basis.

For each thread i , where $1 \leq i \leq n$, let us consider an n -dimensional vector of natural numbers V_i . Intuitively, the number $V_i[j]$ represents the event number at thread t_j that the thread t_i is “aware” of. Since communication in multithreaded systems is done via shared variables, and since reads and writes have different weights in our approach, we let V_x^a and V_x^w be two additional n -dimensional vectors for each shared variable x ; we call the former *access MVC* and the latter *write MVC*. All MVCs are initialized to 0. As usual, for two n -dimensional vectors, $V \leq V'$ if and only if $V[j] \leq V'[j]$ for all $1 \leq j \leq n$, and $V < V'$ if and only if $V \leq V'$ and there is some $1 \leq j \leq n$ such that $V[j] < V'[j]$; also, $\max\{V, V'\}$ is the vector with $\max\{V, V'\}[j] = \max\{V[j], V'[j]\}$ for each $1 \leq j \leq n$.

Our goal is to find a procedure that updates these MVCs and emits a minimal amount of events to an external observer, which can further efficiently extract the relevant causal dependency relation. Formally, the requirements of such a procedure, say \mathcal{A} , which works as an event filter, or an abstraction of the given multithreaded execution, must include the following natural requirements.

Requirements for \mathcal{A} . After \mathcal{A} updates the MVCs as a consequence of the fact that thread t_i generates event e_i^k during the multithreaded execution \mathcal{M} , the following should hold:

- (a) $V_i[j]$ equals the number of relevant events of t_j that causally precede e_i^k ; if $j = i$ and e_i^k is relevant then this number also includes e_i^k ;
- (b) $V_x^a[j]$ equals the number of relevant events of t_j that causally precede the most recent event in \mathcal{M} that accessed (read or wrote) x ; if $i = j$ and e_i^k is a relevant read or write of x event then this number also includes e_i^k ;
- (c) $V_x^w[j]$ equals the number of relevant events of t_j that causally precede the most recent write event of x ; if $i = j$ and e_i^k is a relevant write of x then this number also includes e_i^k .

Finally and most importantly, \mathcal{A} should correctly implement the relative causality (stated formally in Theorem 3).

We next show how such an algorithm can be derived from its requirements above. In order to do it, let us first introduce some useful formal notation. For an event e_i^k of thread t_i , let $(e_i^k]$ be the indexed set $\{(e_i^k)_j\}_{1 \leq j \leq n}$, where $(e_i^k)_j$ is the set $\{e_j^l \mid e_j^l \in t_j, e_j^l \in \mathcal{R}, e_j^l \prec e_i^k\}$ when $j \neq i$ and the set $\{e_i^l \mid l \leq k, e_i^l \in \mathcal{R}\}$ when $j = i$. Intuitively, $(e_i^k]$ contains all the events in the multithreaded computation that causally precede or are equal to e_i^k .

Lemma 1 With the notation above, for $1 \leq i, j \leq n$:

1. $(e_j^{l'})_j \subseteq (e_j^l)_j$ if $l' \leq l$;
2. $(e_j^{l'})_j \cup (e_j^l)_j = (e_j^{\max\{l', l\}})_j$ for any l and l' ;
3. $(e_j^l)_j \subseteq (e_i^k)_j$ for any $e_j^l \in (e_i^k)_j$; and
4. $(e_i^k)_j = (e_j^l)_j$ for some appropriate l .

Proof: 1. is immediate, because for any $l' \leq l$, any event $e_j^{l'}$ at thread t_j preceding or equal to $e_j^{l'}$, that is one with $k \leq l'$, also precedes e_j^l .

2. follows by 1., because it is either the case that $l' \leq l$, in which case $(e_j^{l'})_j \subseteq (e_j^l)_j$, or $l \leq l'$, in which case $(e_j^l)_j \subseteq (e_j^{l'})_j$. In either case 2. holds trivially.

3. There are two cases to analyze. If $i = j$ then $e_j^l \in (e_i^k)_j$ if and only if $l \leq k$, so 3. becomes a special instance of 1.. If $i \neq j$ then by the definition of $(e_i^k)_j$ it follows that $e_j^l \prec e_i^k$. Since $e_j^{l'} \prec e_j^l$ for all $l' < l$ and since \prec is transitive, it follows readily that $(e_j^l)_j \subseteq (e_i^k)_j$.

4. Since $(e_i^k)_j$ is a finite set of totally ordered events, it has a maximum element, say e_i^l . Hence, $(e_i^k)_j \subseteq (e_i^l)_j$. By 3., one also has $(e_i^l)_j \subseteq (e_i^k)_j$. \square

Thus, by 4 above, one can uniquely and unambiguously encode a set $(e_i^k)_j$ by just a number, namely the size of the corresponding set $(e_i^l)_j$, i.e., the number of relevant events of

thread t_j up to its l -th event. This suggests that if the MVC V_i maintained by \mathcal{A} stores that number in its j -th component then (a) in the list of requirements of \mathcal{A} would be fulfilled.

Before we formally show how reads and writes of shared variables affect the causal dependency relation, we need to introduce some notation. First, since a write of a shared variable introduces a causal dependency between the write event and all the previous read or write events of the same shared variable as well as all the events causally preceding those, we need a compact way to refer at any moment to all the read/write events of a shared variable, as well as the events that causally precede them. Second, since a read event introduces a causal dependency to all the previous write events of the same variable as well as all the events causally preceding those, we need a notation to refer to these events as well. Formally, if e_i^k is an event in a multithreaded computation \mathcal{M} and $x \in S$ is a shared variable, then let

$$(e_i^k)_x^a = \begin{cases} \text{The thread-indexed set of all the relevant events that are equal to or causally precede an event } e \text{ accessing } x, \text{ such that } e \text{ occurs before or it is equal to } e_i^k \text{ in } \mathcal{M}, \end{cases}$$

$$(e_i^k)_x^w = \begin{cases} \text{The thread-indexed set of all the relevant events that are equal to or causally precede an event } e \text{ writing } x, \text{ such that } e \text{ occurs before or it is equal to } e_i^k \text{ in } \mathcal{M}. \end{cases}$$

It is obvious that $(e_i^k)_x^w \subseteq (e_i^k)_x^a$. Some or all of the thread-indexed sets of events above may be empty. By convention, if an event, say e , does not exist in \mathcal{M} , then we assume that the indexed sets (e) , $(e)_x^a$, and $(e)_x^w$ are all empty (rather than “undefined”). Note that if \mathcal{A} is implemented such that V_x^a and V_x^w store the corresponding numbers of elements in the index sets of $(e_i^k)_x^a$ and $(e_i^k)_x^w$ immediately after event e_i^k is processed by thread t_i , respectively, then (b) and (c) in the list of requirements for \mathcal{A} are also fulfilled.

Even though the sets of events (e_i^k) , $(e_i^k)_x^a$ and $(e_i^k)_x^w$ have mathematically clean definitions, they are based on total knowledge of the multithreaded computation \mathcal{M} . Unfortunately, \mathcal{M} can be very large in practice, so the computation of these sets may be inefficient if not done properly. Since our analysis algorithms are *online*, we would like to calculate these sets *incrementally*, as the observer receives new events from the instrumented program. A key factor in devising efficient update algorithms is to find equivalent *recursive* definitions of these sets, telling us how to calculate a new set of events from similar sets that have been already calculated at previous event updates.

Let $\{e_i^k\}_i^{\mathcal{R}}$ be the indexed set whose j components are empty for all $j \neq i$ and whose i -th component is either the one element set $\{e_i^k\}$ when $e_i^k \in \mathcal{R}$ or the empty set otherwise. With the notation introduced, the following important

recursive properties hold:

Lemma 2 Let e_i^k be an event in \mathcal{M} and let e_j^l be the event preceding¹ it in \mathcal{M} . If e_i^k is

1. An internal event then

$$\begin{aligned} (e_i^k] &= (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}}, \\ (e_i^k]_x^a &= (e_j^l]_x^a, \text{ for any } x \in S, \\ (e_i^k]_x^w &= (e_j^l]_x^w, \text{ for any } x \in S; \end{aligned}$$

2. A read of x event then

$$\begin{aligned} (e_i^k] &= (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (e_j^l]_x^w, \\ (e_i^k]_x^a &= (e_i^k] \cup (e_j^l]_x^a, \\ (e_i^k]_y^a &= (e_j^l]_y^a, \text{ for any } y \in S \text{ with } y \neq x, \\ (e_i^k]_z^w &= (e_j^l]_z^w, \text{ for any } z \in S; \end{aligned}$$

3. A write of x event then

$$\begin{aligned} (e_i^k] &= (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (e_j^l]_x^a, \\ (e_i^k]_x^a &= (e_i^k], \\ (e_i^k]_x^w &= (e_i^k], \\ (e_i^k]_y^a &= (e_j^l]_y^a, \text{ for any } y \in S \text{ with } y \neq x, \\ (e_i^k]_y^w &= (e_j^l]_y^w, \text{ for any } y \in S \text{ with } y \neq x. \end{aligned}$$

Proof: 1. For the first equality, first recall that $e_i^k \in (e_i^k]$ if and only if e_i^k is relevant. Therefore, it suffices to show that $e \prec e_i^k$ if and only if $e \prec e_i^{k-1}$ for any relevant event $e \in \mathcal{R}$. Since e_i^k is internal, it cannot be in relation \prec_x with any other event for any shared variable $x \in S$, so by the definition of \prec , the only possibilities are that either e is some event $e_i^{k'}$ of thread t_i with $k' < k$, or otherwise there is such an event $e_i^{k'}$ of thread t_i with $k' < k$ such that $e \prec e_i^{k'}$. Hence, it is either the case that e is e_i^{k-1} (so e_i^{k-1} is also relevant) or otherwise $e \prec e_i^{k-1}$. In any of these cases, $e \in (e_i^{k-1}]$. The other two equalities are straightforward consequences of the definitions of $(e_i^k]_x^a$ and $(e_i^k]_x^w$.

2. Like in the proof of 1., $e_i^k \in (e_i^k]$ if and only if $e_i^k \in \mathcal{R}$, so it suffices to show that for any relevant event $e \in \mathcal{R}$, $e \prec e_i^k$ if and only if $e \in (e_i^{k-1}] \cup (e_j^l]_x^w$. Since e_i^k is a read of $x \in S$ event, by the definition of \prec one of the following must hold:

- $e = e_i^{k-1}$. In this case e_i^{k-1} is also relevant, so $e \in (e_i^{k-1}]$;
- $e \prec e_i^{k-1}$. It is obvious in this case that $e \in (e_i^{k-1}]$;
- e is a write of x event and $e \prec_x e_i^k$. In this case $e \in (e_j^l]_x^w$;
- There is some write of x event e' such that $e \prec e'$ and $e' \prec_x e_i^k$. In this case $e \in (e_j^l]_x^w$, too.

¹If e_i^k is the first event then we can assume that e_j^l does not exist in \mathcal{M} , so by convention all the associated sets of events are empty

Therefore, $e \in (e_i^{k-1}]$ or $e \in (e_j^l]_x^w$.

Let us now prove the second equality. By the definition of $(e_i^k]_x^a$, one has that $e \in (e_i^k]_x^a$ if and only if e is equal to or causally precedes an event accessing $x \in S$ that occurs before or is equal to e_i^k in \mathcal{M} . Since e_i^k is a read of x , the above is equivalent to saying that either it is the case that e is equal to or causally precedes e_i^k , or it is the case that e is equal to or causally precedes an event accessing x that occurs *strictly before* e_i^k in \mathcal{M} . Formally, the above is equivalent to saying that either $e \in (e_i^k]$ or $e \in (e_j^l]_x^a$. If $y, z \in S$ and $y \neq x$ then one can readily see (like in 1. above) that $(e_i^k]_y = (e_j^l]_y$ and $(e_i^k]_z = (e_j^l]_z$.

3. It suffices to show that for any relevant event $e \in \mathcal{R}$, $e \prec e_i^k$ if and only if $e \in (e_i^{k-1}] \cup (e_j^l]_x^a$. Since e_i^k is a write of $x \in S$ event, by the definition of \prec one of the following must hold:

- $e = e_i^{k-1}$. In this case $e_i^{k-1} \in \mathcal{R}$, so $e \in (e_i^{k-1}]$;
- $e \prec e_i^{k-1}$. It is obvious in this case that $e \in (e_i^{k-1}]$;
- e is an access of x event (read or write) and $e \prec_x e_i^k$. In this case $e \in (e_j^l]_x^a$;
- There is some access of x event e' such that $e \prec e'$ and $e' \prec_x e_i^k$. In this case $e \in (e_j^l]_x^a$, too.

Therefore, $e \in (e_i^{k-1}]$ or $e \in (e_j^l]_x^a$.

For the second equality, note that, as for the second equation in 2., one can readily see that $e \in (e_i^k]_x^a$ if and only if $e \in (e_i^k] \cup (e_j^l]_x^a$. But $e_j^l]_x^a \subseteq (e_i^k]$, so the above is equivalent to $e \in (e_i^k]$. A similar reasoning leads to $(e_i^k]_x^w = (e_i^k]$. The equalities for $y \neq x$ immediate, because e_i^k has no relation to accesses of other shared variables but x . \square

Since each component set of each of the indexed sets in these recurrences has the form $(e_i^k)_i$ for appropriate i and k , and since each $(e_i^k)_i$ can be safely encoded by its size, one can then safely encode each of the above indexed sets by an n -dimensional MVC; these MVCs are precisely V_i for all $1 \leq i \leq n$ and V_x^a and V_x^w for all $x \in S$. It is a simple exercise now to derive the following MVC update algorithm \mathcal{A} (which was also given in Section 1):

ALGORITHM \mathcal{A}

INPUT: event e generated by thread t_i

1. if e is relevant then
 $V_i[i] \leftarrow V_i[i] + 1$
2. if e is a read of a shared variable x then
 $V_i \leftarrow \max\{V_i, V_x^w\}$
 $V_x^a \leftarrow \max\{V_x^a, V_i\}$
3. if e is a write of a shared variable x then
 $V_x^w \leftarrow V_x^a \leftarrow V_i \leftarrow \max\{V_x^a, V_i\}$

4. if e is relevant then
send message $\langle e, i, V_i \rangle$ to observer

An interesting observation is that one can regard the problem of recursively calculating (e_i^k) as a dynamic programming problem. As can often be done in dynamic programming problems, one can reuse space and derive the Algorithm \mathcal{A} . Therefore, \mathcal{A} satisfies all the stated requirements (a), (b) and (c), so they can be used as properties next in order to show the correctness of \mathcal{A} :

Theorem 3 *If $\langle e, i, V \rangle$ and $\langle e', i', V' \rangle$ are two messages sent by \mathcal{A} , then $e \triangleleft e'$ if and only if $V[i] \leq V'[i]$ (no typo: the second i is not an i') if and only if $V < V'$.*

Proof: First, note that e and e' are both relevant. The case $i = i'$ is trivial. Suppose $i \neq i'$. Since, by requirement (a) for \mathcal{A} , $V[i]$ is the number of relevant events that t_i generated before and including e and since $V'[i]$ is the number of relevant events of t_i that causally precede e' , it is clear that $V[i] \leq V'[i]$ if and only if $e \prec e'$. For the second part, if $e \triangleleft e'$ then $V \leq V'$ follows again by requirement (a), because any event that causally precedes e also precedes e' . Since there are some indices i and i' such that e was generated by t_i and e' by $t_{i'}$, and since $e' \not\prec e$, by the first part of the theorem it follows that $V'[i'] > V[i']$; therefore, $V < V'$. For the other implication, if $V < V'$ then $V[i] \leq V'[i]$, so the result follows by the first part of the theorem. \square

3.1. Synchronization and Shared Variables

Thread communication in multithreaded systems was considered so far to be accomplished by writing/reading shared variables, which were assumed to be known *a priori*. In the context of a language like Java, this assumption works only if the shared variables are declared *static*; it is less intuitive when synchronization and dynamically shared variables are considered as well. Here we show that, under proper instrumentation, the basic algorithm presented in the previous subsection also works in the context of synchronization statements and dynamically shared variables.

Since in Java synchronized blocks cannot be interleaved, so corresponding events cannot be permuted, locks are considered as shared variables and a write event is generated whenever a lock is acquired or released. This way, a causal dependency is generated between any exit and any entry of a synchronized block, namely the expected happens-before relation. Java synchronization statements are handled exactly the same way, that is, the shared variable associated to the synchronization object is written at the entrance and at the exit of the synchronized region. Condition synchronizations (wait/notify) can be handled similarly, by generating a write of a dummy shared variable by both the notifying

thread before notification and by the notified thread after notification.

To handle variables that are dynamically shared, for each variable x of primitive type in each class the instrumentation program adds *access* and *write* MVCs, namely *_access_mvc_x* and *_write_mvc_x*, as new fields in the class. Moreover, for each read and write access of a variable of primitive type in any class, it adds codes to update the MVCs according to the multithreaded vector clock algorithm.

3.2. A Distributed Systems Interpretation

It is known that the various mechanisms for process interaction are essentially equivalent. This leads to the following natural question: could it be possible to derive the MVC algorithm in this section from vector clock based algorithms implementing causality in distributed systems, such as the ones in [2, 7]. The answer to this question is: *almost*.

Since writes and accesses of shared variables have different impacts on the causal dependency relation, the most natural thing to do is to associate two processes to each shared variable x , one for accesses, say x^a and one for writes, say x^w . As shown in Fig. 3 right, a write of x by thread i can be seen as sending a “request” message to write x to the “access process” x^a , which further sends a “request” message to the “write process” x^w , which performs the action and then sends an acknowledgment message back to i . This is consistent with step 3 of the algorithm in Fig. 2; to see this, note that $V_x^w \leq V_x^a$ at any time.

However, a read of x is less obvious and does not seem to be interpretable by message passing updating the MVCs the standard way. The problem here is that the MVC of x^a needs to be updated with the MVC of the accessing thread i , the MVC of the accessing thread i needs to be updated with the current MVC of x^w in order to implant causal dependencies between previous writes of x and the current access, but the point here is that the MVC of x^w does not

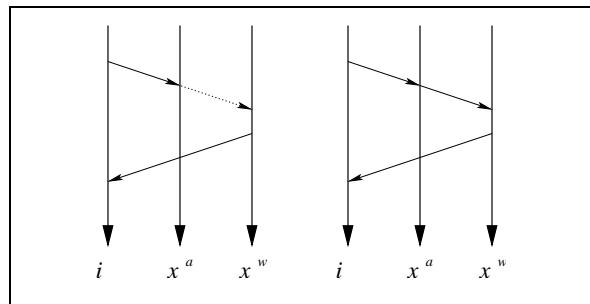


Figure 3. A distributed systems interpretation of reads (left) and writes (right).

have to be updated by reads of x ; this is what allows reads to be permutable by the observer. In terms of message passing, like Fig. 3 shows, this says that the access process x^a sends a *hidden* request message to x_w (after receiving the read request from i), whose only role is to “ask” x^w send an acknowledgment message to i . By hidden message, marked with dotted line in Fig. 3, we mean a message which is not considered by the standard MVC update algorithm. The role of the acknowledgment message is to ensure that i updates its MVC with the one of the write access process x^w .

4. The Vector Clock Algorithm at Work

In this section we propose predictive runtime analysis frameworks in which the presented MVC algorithm can be used, and describe by examples how we use it in JAVA MULTIPATHEXPLORER (JMPAX) [27, 28, 19].

The observer therefore receives messages of the form $\langle e, i, V \rangle$ in any order, and, thanks to Theorem 3, can extract the causal partial order \triangleleft on relevant events, which is its abstraction of the running program. Any permutation of the relevant events which is consistent with \triangleleft is called a *multithreaded run*, or simply a *run*. Notice that each run corresponds to some possible execution of the program under different execution speeds or scheduling of threads, and that the observed sequence of events is just one such run. Since each relevant event contains global state update information, each run generates a sequence of global states. If one puts all these sequences together then one gets a lattice, called *computation lattice*. The reader is assumed familiar with techniques on how to extract a computation lattice from a causal order given by means of vector clocks [24]. Given a global property to analyze, the task of the observer now is to verify it against every path in the automatically extracted computation lattice. JPAX and JAVA-MAC are able to analyze only one path in the lattice. The power of our technique consists of its ability to predict potential errors in other possible multithreaded runs.

Once a computation lattice containing all possible runs is extracted, one can start using standard techniques on debugging distributed systems, considering both state predicates [29, 7, 5] and more complex, such as temporal, properties [3, 5, 1, 4]. Also, the presented algorithm can be used as a front-end to partial order trace analyzers such as POTA [26]. Also, since the computation lattice acts like an abstract model of the running program, one can potentially run one’s favorite model checker against any property of interest. We think, however, that one can do better than that if one takes advantage of the specific runtime setting of the proposed approach. The problem is that the computation lattice can grow quite large, in which case storing it might become a significant matter. Since events are received incrementally from the instrumented program, one can buffer them at the

observer’s side and then build the lattice on a level-by-level basis in a top-down manner, as the events become available. The observer’s analysis process can also be performed incrementally, so that parts of the lattice which become non-relevant for the property to check can be garbage-collected while the analysis process continues.

If the property to be checked can be translated into a finite state machine (FSM) or if one can synthesize online monitors for it, like we did for safety properties [28, 17, 18, 27], then one can analyze all the multithreaded runs *in parallel*, as the computation lattice is built. The idea is to store the state of the FSM or of the synthesized monitor together with each global state in the computation lattice. This way, in any global state, all the information needed about the past can be stored via a set of states in the FSM or the monitor associated to the property to check, which is typically quite small in comparison to the computation lattice. Thus only one cut in the computation lattice is needed at any time, in particular one level, which significantly reduces the space required by the proposed predictive analysis algorithm.

Liveness properties apparently do not fit our runtime verification setting. However, stimulated by recent encouraging results in [22], we believe that it is also worth exploring techniques that can *predict violations of liveness properties*. The idea here is to search for paths of the form uv in the computation lattice with the property that the shared variable global state of the multithreaded program reached by u is the same as the one reached by uv , and then to check whether uv^ω satisfies the liveness property. The intuition here is that the system can potentially run into the infinite sequence of states uv^ω (u followed by infinity many repetitions of v), which may violate the liveness property. It is shown in [22] that the test $uv^\omega \models \varphi$ can be done in polynomial time and space in the sizes of u , v and φ , typically linear in uv , for almost any temporal logic.

4.1. Java MultiPathExplorer (JMPaX)

JMPAX [27, 28] is a runtime verification tool which checks a user defined specification against a running program. The specifications supported by JMPAX allow any temporal logic formula, using an interval-based notation built on state predicates, so our properties can refer to the entire history of states. Fig. 4 shows the architecture of JMPAX. An instrumentation module parses the user specification, extracts the set of shared variables it refers to, i.e., the relevant variables, and then *instruments* the multithreaded program (which is assumed in bytecode form) as follows. Whenever a shared variable is accessed the MVC algorithm \mathcal{A} in Section 3 is inserted; if the shared variable is relevant and the access is a write then the event is considered relevant. When the instrumented bytecode is executed, mes-

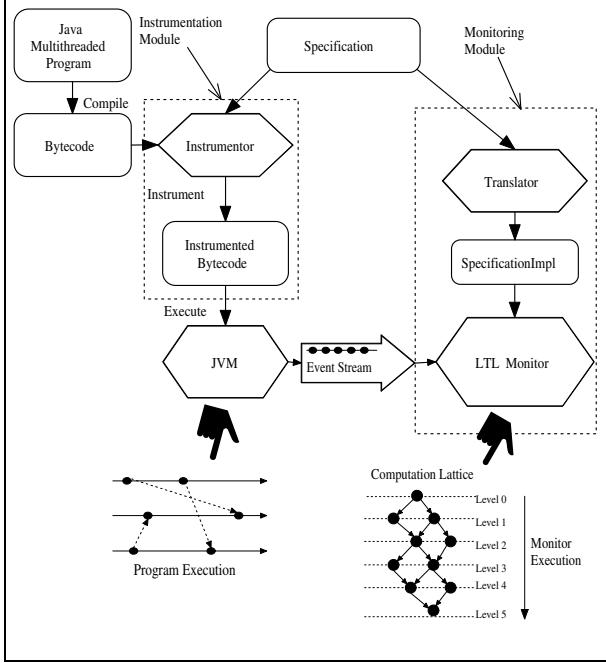


Figure 4. The Architecture of JMPAX.

sages $\langle e, i, V \rangle$ for relevant events e are sent via a socket to an external observer.

The observer generates the computation lattice on a level by level basis, checking the user defined specification against all possible multithreaded runs in parallel. Note that only one of those runs was indeed executed by the instrumented multithreaded program, and that the observer does not know it; the other runs are *potential* runs, they can occur in other executions of the program. Despite the exponential number of potential runs, at most two consecutive levels in the computation lattice need to be stored at any moment. [27, 28] gives more details on the particular implementation of JMPAX. We next discuss two examples where JMPAX can predict safety violations from successful runs; the probability of detecting these bugs only by monitoring the observed run, as JPAX and JAVA-MAC do, is very low.

Example 1. Let us consider the simple landing controller in Fig.1, together with the property “If the plane has started landing, then it is the case that landing has been approved and since then the radio signal has never been down.” Suppose that a successful execution is observed, in which the radio goes down *after* the landing has started. After instrumentation, this execution emits only three events to the observer in this order: a write of `approved` to 1, a write of `landing` to 1, and a write of `radio` to 0. The observer can now build the lattice in Fig.5, in which the states are encoded by triples $\langle \text{landing}, \text{approved}, \text{radio} \rangle$ and the

leftmost path corresponds to the observed execution. However, the lattice contains two other runs both violating the safety property. The rightmost one corresponds to the sit-

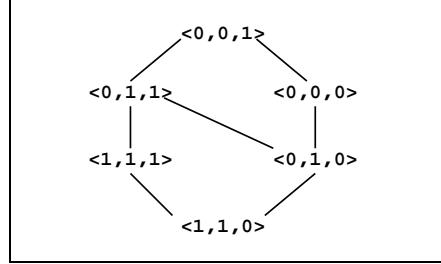


Figure 5. Computation lattice for the program in Fig. 1.

uation when the radio goes down right between the test `radio==0` and the action `approved=1`, and the inner one corresponds to that in which the radio goes down between the actions `approved=1` and `landing=1`. Both these erroneous behaviors are insightful and very hard to find by testing. JMPAX is able to build the two counterexamples very quickly, since there are only 6 states to analyze and three corresponding runs, so it is able to give useful feedback.

Example 2. Let us now consider an artificial example intended to further clarify the prediction technique. Suppose that one wants to monitor the safety property “if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false” against a multithreaded program in which initially $x = -1$, $y = 0$ and $z = 0$, with one thread containing the code $x++$; ...; $y = x + 1$ and another containing $z = x + 1$; ...; $x++$. The dots indicate code that is not relevant, i.e., that does not access the variables x , y and z . This multithreaded program, after instrumentation, sends messages to JMPAX’s observer whenever the relevant variables x, y, z are updated. A possible execution of the program to be sent to the observer can consist of the sequence of program states $(-1, 0, 0)$, $(0, 0, 0)$, $(0, 0, 1)$, $(1, 0, 1)$, $(1, 1, 1)$, where the tuple $(-1, 0, 0)$ denotes the state in which $x = -1$, $y = 0$, $z = 0$. Following the MVC algorithm, we can deduce that the observer will receive the multithreaded computation shown in Fig. 6, which generates the computation lattice shown in the same figure. Notice that the observed multithreaded execution corresponds to just one particular multithreaded run out of the three possible, namely the leftmost one. However, another possible run of the same computation is the rightmost one, which violates the safety property. Systems like JPAX and JAVA-MAC that analyze only the observed runs fail to detect this violation. JMPAX predicts this bug from the original successful run.

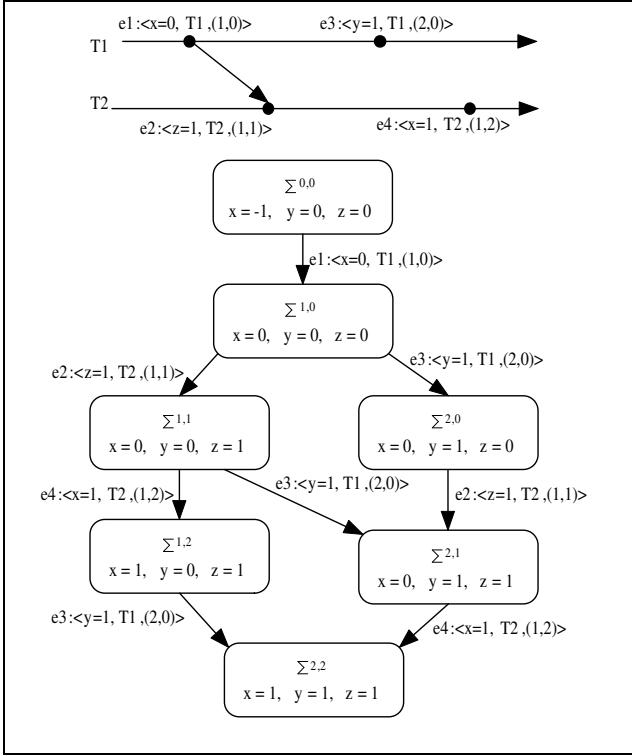


Figure 6. Computation lattice with three runs.

5. Conclusion

A simple and effective algorithm for extracting the relevant causal dependency relation from a running multithreaded program was presented in this paper. This algorithm is supported by JMPAX, a runtime verification tool able to detect and predict safety errors in multithreaded programs.

Acknowledgments. Many thanks to Gul Agha and Mark-Oliver Stehr for their inspiring suggestions and comments on several previous drafts of this work. The work is supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586, the DARPA IXO NEST Program, contract number F33615-01-C-1907), the ONR Grant N00014-02-1-0715, the Motorola Grant MOTOROLA RPS #23 ANT, and the joint NSF/NASA grant CCR-0234524.

References

- [1] M. Ahamad, M. Raynal, and G. Thia-Kime. An adaptive protocol for implementing causally consistent distributed services. In *Proceedings of International Conference on Distributed Computing (ICDCS'98)*, pages 86–93, 1998.
- [2] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.
- [3] O. Babaoglu and M. Raynal. Specification and verification of dynamic properties in distributed computations. *Journal of Parallel and Distr. Computing*, 28(2):173–185, 1995.
- [4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings Verification, Model Checking and Abstract Interpretation (VMCAI 04)*, volume 2937 of *LNCS*, pages 44–57, Jan. 2004.
- [5] C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.
- [6] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.
- [7] R. Cooper and K. Marzullo. Consistent detection of global predicates. *ACM SIGPLAN Notices*, 26(12):167–174, 1991. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.
- [8] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of Formal Methods Europe (FME'93): Industrial Strength Formal Methods*, volume 670 of *LNCS*, pages 268–284, 1993.
- [9] D. Drusinsky. Temporal rover. <http://www.time-rover.com>.
- [10] D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proc. of SPIN'00: SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330, Stanford, California, USA, 2000. Springer.
- [11] D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *Proc. of CAV'03: Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 114–118, Boulder, Colorado, USA, 2003. Springer-Verlag.
- [12] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN/SIGOPS workshop on Parallel and Distr. Debugging*, pages 183–194. ACM, 1988.
- [13] E. L. Gunter, R. P. Kurshan, and D. Peled. PET: An interactive software testing tool. In *Computer Aided Verification (CAV'00)*, volume 1885 of *LNCS*, pages 552–556. Springer-Verlag, 2003.
- [14] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier, 2001.
- [15] K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings Automated Software Engineering (ASE'01)*, pages 135–143. IEEE, 2001.
- [16] K. Havelund and G. Roşu. *Runtime Verification 2001, 2002*, volume 55, 70(4) of *ENTCS*. Elsevier, 2001, 2002. Proceedings of a *Computer Aided Verification (CAV'01, CAV'02)* workshop.
- [17] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools and Tech. Transfer*, to appear.
- [18] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.

- [19] Java MultiPathExplorer. <http://fsl.cs.uiuc.edu/jmpax>.
- [20] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier Science, 2001.
- [21] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
- [22] N. Markey and P. Schnoebelen. Model checking a path (preliminary report). In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR'2003)*, volume 2761 of *LNCS*, pages 248–262. Springer, 2003.
- [23] K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WADG'91)*, volume 579 of *LNCS*, pages 254–272. Springer, 1991.
- [24] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1989.
- [25] G. Roşu and K. Sen. An instrumentation technique for online analysis of multithreaded programs. In *PADTAD workshop at IPDPS*. IEEE Computer Society, 2003.
- [26] A. Sen and V. K. .Garg. Partial order trace analyzer (pota) for distributed programs. In *Proceedings of Workshop on Runtime Verification (RV'03)*, ENTCS, 2003.
- [27] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*. ACM, 2003.
- [28] K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 123–138, Barcelona, Spain, Mar. 2004. Springer.
- [29] S. D. Stoller. Detecting global predicates in distributed systems with clocks. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, pages 185–199, 1997.
- [30] S. A. Vilkomir and J. P. Bowen. Formalization of software testing criteria using the Z notation. In *Proceedings of COMPSAC 01: 25th IEEE Annual International Computer Software and Applications Conference*, pages 351–356. IEEE Computer Society, Oct. 2001.

ON PARALLEL vs. SEQUENTIAL THRESHOLD CELLULAR AUTOMATA

PREDRAG TOSIC* and GUL AGHA

*Open Systems Laboratory (<http://osl.cs.uiuc.edu>), Department of Computer Science,
University of Illinois at Urbana-Champaign, 201 N. Goodwin, Urbana, IL 61801, USA
[{p-tosic, agha}](mailto:{p-tosic, agha}@cs.uiuc.edu)@cs.uiuc.edu*

ABSTRACT

Cellular automata (CA) are an abstract model of fine-grain parallelism, as the node update operations are rather simple, and therefore comparable to the basic operations of the computer hardware. In a classical CA, all the nodes execute their operations in parallel and in perfect synchrony. We consider herewith the sequential version of CA, called SCA, and compare these SCA with the classical, parallel CA. In particular, we show that there are 1-D CA with very simple node state update rules that cannot be simulated by any comparable SCA, irrespective of the node update ordering. Consequently, the fine granularity of the basic CA operations and, therefore, the fine-grain parallelism of the classical, synchronous CA, insofar as the “interleaving semantics” is concerned, is not fine enough. We also share some thoughts on how to extend the results herein, and, in particular, we try to motivate the study of genuinely asynchronous cellular automata.

Keywords: cellular automata, linear threshold automata, dynamical systems, concurrency, sequential interleaving semantics

1. Introduction and Motivation

Cellular automata (CA) were originally introduced as an abstract mathematical model that can capture the behavior of biological systems capable of self-reproduction [19]. Subsequently, CA have been extensively studied in a great variety of application domains, mostly in the context of complex physical or biological systems and their dynamics (e.g., [11, 26, 27, 28, 29]). However, CA can also be viewed as an abstraction of massively parallel computers (e.g, [8]). Herein, we study a particular simple yet nontrivial class of CA from the parallel and distributed computing perspectives. In particular, we pose - and partially answer - some fundamental questions regarding the nature of the CA parallelism, i.e., the perfect synchrony of the classical CA computation.

It is well known that CA are an abstract architecture model of *fine-grain parallelism*, in that the elementary operations executed at each node are rather simple and hence comparable to the basic operations performed by the computer hardware. In a classical (parallel) CA, whether finite or infinite, all the nodes execute their operations in parallel and *in perfect synchrony*, that is, *logically simultaneously*: in general, the state of a node x_i at time step $t + 1$ is some simple function of the states of the node x_i and a set of its pre-specified neighbors at time t .

We consider herewith the sequential version of CA, that we shall abridge to SCA in the sequel, and compare these SCA with the perfectly synchronous *parallel* (or *concurrent*) CA. In particular, we will show that there are 1-D CA with very simple state update rules that cannot be simulated by any comparable SCA, irrespective of the node update ordering. While the result would be trivial if one considers a single (S)CA computation, we argue that the result is nontrivial and important when applied to *all possible inputs* (starting configurations) and, moreover, to the entire classes of CA and SCA. Hence, the granularity of the basic CA operations, insofar as the (im)possibility of simulating

*Contact author. Work phone: 217-244-1976. Fax: 217-333-9386.

their concurrent computation via appropriate sequential interleavings of these basic operations, turns out not to be quite *fine enough*. We also share some thoughts on how to extend the results presented herein, and, in particular, we try to motivate the study of *genuinely asynchronous cellular automata*, where the asynchrony applies not only to the local computations at individual nodes, but also to the *communication* among different nodes via the “shared variables” stored as the respective nodes’ states.

An example of asynchrony in the local node updates (i.e., asynchronous computation at different “processors”) is when, for instance, the individual nodes update one at a time, according to some random order. This is a kind of asynchrony found in the literature, e.g., in [14, 15]. It is important to understand, however, that even in case of what is referred to as *asynchronous cellular automata (ACA)* in the literature, the term *asynchrony* there applies to local updates (i.e., computations) *only*, but not to communication, since a tacit assumption of the globally accessible global clock still holds. We prefer to refer to this kind of (weakly asynchronous) *(A)CA* as *sequential cellular automata*, and, in this work, consistently keep the term *asynchronous cellular automata* for those *CA* that do not have a global clock (see *Section 4*).

Before dwelling into the issue of concurrency vs. arbitrary sequential interleavings applied to the threshold cellular automata, we first clarify the terminology, and then introduce the relevant concepts through a simple programming exercise in *Subsection 1.1*.

Throughout, we use the terms *parallel* and *concurrent* as synonyms. Many programming languages experts would strongly disagree with this convention. However, a complete agreement in the computer science community on what exactly *concurrency* means, and how it relates to *parallelism*, is lacking. According to *Chapter §12* of [22], “concurrency in the programming language and parallelism in the computer hardware are independent concepts. [...] We can have concurrency in a programming language without parallel hardware, and we can have parallel execution without concurrency in the language. In short, *concurrency refers to the potential for parallelism*” (italics ours). Clearly, our convention herein does not conform to the notions of concurrency and parallelism as defined in [22]. In contrast, [20] uses the term *concurrent* “to describe computations where the simultaneously executing processes can interact with one another”, and *parallel* for “[...] computations where behavior of each process is unaffected by the behavior of the others”. [20] also acknowledges that many authors do not discriminate between “*parallel*” and “*concurrent*”. We shall follow this latter convention throughout and, moreover, by a *parallel (concurrent) computation* we shall mean actions of several processing units that are carried out *logically* (if not necessarily *physically*) *simultaneously*. That is, when referring to parallel (or, equivalently, concurrent) computation, we shall always assume a *perfect synchrony*.

1.1. Capturing Concurrency by Sequential Interleavings

While our own brains are massively parallel computing devices, we seem to (consciously) think and approach problem-solving rather sequentially. In particular, when designing a parallel algorithm or writing a computer program that is inherently parallel, we still prefer to be able to understand such an algorithm or program at the level of sequential operations or executions. It is not surprising, therefore, that a great deal of research effort has been devoted to interpreting parallel computation in the more familiar, sequential terms. One of the most important contributions in that respect is the (nondeterministic) sequential *interleaving semantics* of concurrency (see, e.g., [7, 9, 13, 17, 18]).

When interpreting concurrency via interleaving semantics, a natural question arises: *Given a parallel computing model, can its parallel execution always be captured by such sequential nondeterminism, so that any given parallel computation can be faithfully reproduced via an appropriate choice of a sequential interleaving of the operations involved?* For most theoreticians of parallel computing^a,

^aThat is, for all “believers” in the interleaving semantics of concurrency - as contrasted with, e.g., proponents of

the answer is apparently “Yes” - provided that we simulate concurrent execution via sequential interleavings at a sufficiently high level of granularity of the basic computational operations. However, given a parallel computation in the form of a set of concurrently executing processes, how do we tell if the particular level of granularity is *fine enough*, i.e., whether the operations at that granularity level can truly be rendered *atomic* for the purposes of capturing concurrency via sequential interleavings?

We shall illustrate the concept of *sequential interleaving semantics* of concurrency with a simple example. Let’s consider the following trivia question from a sophomore parallel programming class: *Find a simple example of two instructions such that, when executed in parallel, they give a result not obtainable from any corresponding sequential execution sequence?*

A possible answer: Assume $x = 0$ initially and consider the following two programs

$x \leftarrow x + 1; x \leftarrow x + 1$

vs.

$x \leftarrow x + 1 \parallel x \leftarrow x + 1$

where “ \parallel ” stands for the parallel, and “ $;$ ” for the sequential composition of instructions or programs, respectively. Sequentially, one *always* gets the same answer: $x = 2$. In parallel (when the two assignment operations are executed synchronously), however, one gets $x = 1$. It appears, therefore, that no sequential ordering of operations can reproduce parallel computation - at least not at the granularity level of high-level instructions as above.

The whole “mystery” can be readily resolved if we look at the possible sequential executions of the corresponding machine instructions:

LOAD $x, *m$	LOAD $x, *m$
ADD $x, \#1$	ADD $x, \#1$
STORE $x, *m$	STORE $x, *m$

There certainly exist choices of *sequential interleavings* of the six machine instructions above that lead to “*parallel*” behavior (i.e., the one where, after the code is executed, $x = 1$). Thus, by refining granularity from the high-level language instructions down to the machine instructions, we can certainly preserve the interleaving “semantics” of concurrency.

As a side, we remark that it turns out that the example above does not require finer granularity quite yet, if we allow that some instructions be treated as no-ops. Indeed, if we informally define $\Phi(P)$ to be the *set of possible behaviors of program P*, then the example above only shows that, for $S_1 = S_2 = (x \leftarrow x + 1)$,

$$\Phi(S_1 \parallel S_2) \not\subseteq \Phi(S_1; S_2) \cup \Phi(S_2; S_1) \quad (1)$$

However, it turns out that, in this particular example, it indeed is the case that

$$\Phi(S_1 \parallel S_2) \subseteq \Phi(S_1; S_2) \cup \Phi(S_2; S_1) \cup \Phi(S_1) \cup \Phi(S_2) \quad (2)$$

and no finer granularity is necessary to model $\Phi(S_1 \parallel S_2)$, assuming that, in some of the sequential interleavings, we allow certain instructions not to be executed at all.

However, one can construct more elaborate examples where the property (2) does not hold. The only way to capture the program behavior of parallel compositions of the form $\Phi(P_1 \parallel P_2)$ via sequential interleavings in such cases would then be to find a finer level of granularity, i.e., to reconsider at what level can operations be considered *atomic*, so that the union of all possible sequential interleavings of such basic operations (including the interleavings that allow “no-ops” for some of the instructions) is guaranteed to capture the concurrent behavior, i.e., so that (2) holds. That is, sometimes *refining the granularity of operations* so that sequential interleavings can capture synchronous parallel behavior, becomes a *necessity*.

true concurrency, an alternative model not discussed herewith.

We address herein the (in)adequacy of the sequential interleavings semantics when applied to *CA* where the individual node updates^b are considered to be elementary operations. In particular, we show that the perfect synchrony of the classical *CA*'s node updates causes the interleaving semantics, as captured by the *SCA* and *NICA* sequential CA models (*Section 2*), to fail rather dramatically even in the context of the simplest (nonlinear) CA node update rules.

2. Cellular Automata and Types of Their Configurations

We introduce *CA* by first considering (*deterministic*) *Finite State Machines (FSMs)* such as *Deterministic Finite Automata (DFA)*. An *FSM* has finitely many states, and is capable of reading the input signals coming from the outside. The machine is initially in some starting state; upon reading each input signal, the machine changes its state according to a pre-defined and fixed rule. In particular, the entire memory of the system is contained in what “current state” the machine is in, and nothing else about the previously processed inputs is remembered. Hence, the probabilistic generalization of deterministic *FSMs* leads to (discrete) Markov chains. It is important to notice that there is no way for a *FSM* to overwrite, or in any other way affect, the input data stream. Thus *individual FSMs* are computational devices of rather limited power.

Now let us consider many such *FSMs*, all identical to one another, that are lined up together in some regular fashion, e.g., on a straight line or a regular 2-D grid, so that each single “node” in the grid is connected to its immediate neighbors. Let's also eliminate any external sources of input streams to the individual machines at the nodes, and let the current values of any given node's neighbors be that node's only “input data”. If we then specify a finite set of the possible values held in each node, and we also identify this set of values with the set of each node's *internal states*, we arrive at an informal definition of a classical cellular automaton. To summarize, a *CA* is a finite or infinite regular grid in one-, two- or higher-dimensional space, where each node in the grid is a *FSM*, and where each such node's input data at each time step are the corresponding internal states of the node's neighbors. Moreover, in the most important special case - the Boolean case, this *FSM* is particularly simple, i.e., it has only two possible internal states, labeled 0 and 1. All the nodes of a classical *CA* execute the *FSM* computation in unison, i.e., (*logically*) *simultaneously*. We note that infinite *CA* are capable of universal (Turing) computation. Moreover, the general class of infinite *CA*, once arbitrary starting configurations are allowed, are actually strictly more powerful than the classical Turing machines (for more, see, e.g., [8]).

We follow [8] and formally define classical (that is, synchronous and concurrent) *CA* in two steps: we first define the notion of a *cellular space*, and, subsequently, we define a *cellular automaton* over an appropriate cellular space.

Definition 1 A *Cellular Space*, Γ , is an ordered pair (G, Q) , where

- G is a regular undirected Cayley graph that may be finite or infinite, with each node labeled with a distinct integer; and
- Q is a finite set of states that has at least two elements, one of which being the special *quiescent state*, denoted by 0.

We denote the set of integer labels of the nodes (vertices) in Γ by L . That is, L may be equal to, or be a proper subset of, the set of all integers.

^bIt is tacitly assumed here that the complete node update operation includes, in addition to computing the local update function on appropriate inputs, also the necessary *reads* of the neighbors' values preceding the local rule computation, as well as the *writes* of one's new value following the local computation. These points will become clear once the necessary definitions and terminology are introduced in *Section 2*; see also discussion in *Sections 4 and 5*.

Definition 2 A *Cellular Automaton* A is an ordered triple (Γ, N, M) , where

- Γ is a *cellular space*;
- N is a *fundamental neighborhood*; and
- M is a *finite state machine* such that the input alphabet of M is $Q^{|N|}$, and the local transition function (update rule) for each node is of the form $\delta : Q^{|N|+1} \rightarrow Q$ for *CA with memory*, and $\delta : Q^{|N|} \rightarrow Q$ for *memoryless CA*.

The fundamental neighborhood N specifies what near-by nodes provide inputs to the update rule of a given node. In the classical *CA*, Γ is a regular graph that locally “looks the same everywhere”; in particular, the local neighborhood N is the same for each node in Γ .

The local transition rule δ specifies how each node updates its state (that is, value), based on its current state (value), and the current states of its neighbors in N . By composing together the application of the local transition rule to each of the *CA*’s nodes, we obtain *the global map* on the set of (global) configurations of a cellular automaton.

We observe that there is plenty of parallelism in the *CA* “hardware”, assuming, of course, a sufficiently large number of nodes^c. Actually, classical *CA* defined over infinite cellular spaces provide *unbounded parallelism* where, in particular, an infinite amount of information processing is carried out in a finite time (even in a single parallel step). In particular, the notion of independence between parallelism and concurrency as defined in [22] seems inappropriate to apply to *CA*: without the parallel “hardware”, that is, multiple interconnected nodes, a *CA* is not capable of *any* concurrent computation. Indeed, a single-node *CA* is just a “fixed” deterministic *FSM* - an entirely sequential computing model.

Insofar as the *CA* “computer architecture” is concerned, one important characteristic is that the memory and the processors are not truly distinguishable, in stark contrast to *Turing machines*, *(P)RAMs*, and other standard abstract models of digital computers. Namely, each node of a cellular automaton is both a processing unit and a memory storage unit; see, e.g., the detailed discussion in [24]. In particular, the only “memory content” of a *CA* is a tuple of the (current) states of all its nodes. Moreover, as a node can “read” (but not “write”) the states or “values” of its neighbors, we can view the architecture of classical *CA* as a very simplistic, special case of *distributed shared memory* parallel model, where every “processor” (that is, each node) “owns” one cell (typically, one bit) of its “local memory”, physically separated from other similar local “memories” - yet this local memory is *directly accessible* (for *read* accesses) to some of the other “processors”. In particular, the “reads” to any “memory cell” (or a “shared variable” stored in such a memory cell) are restricted to an appropriate neighborhood of that shared value’s “owner processor”, while the “writes” are restricted to the owner processor *alone*.

Since our main results herein pertain to a comparison and contrast between the classical, concurrent threshold *CA* and their sequential counterparts, we formally introduce two types of the sequential *CA* next. First, we define *SCA* with a *fixed* (but arbitrary) sequence specifying the order according to which the nodes are to update. We then introduce a kind of sequential automata whose purpose is to capture the “interleaving semantics”, that is, where *all* possible sequences of node updates are considered at once.

Definition 3 A *Sequential Cellular Automaton (SCA)* S is an ordered quadruple (Γ, N, M, s) , where Γ , N and M are as in *Definition 2*, and s is an arbitrary sequence, finite or infinite, all of whose elements are drawn from the set L of integers used in labeling the vertices of Γ . The sequence s is specifying the sequential ordering according to which an *SCA*’s nodes update their states, one at a time.

^cSee the discussion in *Section 1*, and, in particular, the definition of the relationship between concurrency and parallelism in reference [22].

However, when comparing and contrasting the concurrent *CA* with their sequential counterparts, rather than making a comparison between a given *CA* with a *particular SCA* (that is, a corresponding *SCA* with some particular choice of the update sequence s), we compare the parallel *CA* computations with the computations of the corresponding *SCA* for *all* possible sequences of node updates. For that purpose, the following class of sequential automata is introduced:

Definition 4 A *Nondeterministic Interleavings Cellular Automaton (NICA)* I is defined to be the union of all sequential automata S whose first three components, Γ, N and M are fixed. That is, $I = \cup_s (\Gamma, N, M, s)$, where the meanings of Γ, N, M , and s are the same as before, and the union is taken over *all* (finite and infinite) sequences $s : \{1, 2, 3, \dots\} \rightarrow L$, where L is the set of integer labels of the nodes in Γ .

We now change pace and introduce some terminology from physics that we find useful for characterizing *all possible computations* of a parallel or a sequential cellular automaton. To this end, a (*discrete*) *dynamical system* view of *CA* is helpful. A *phase space* of a dynamical system is a directed graph where the vertices are the *global configurations* (or *global states*) of the system, and directed edges correspond to the possible direct transitions from one global state to another.

As for any other kind of dynamical systems, we can define the fundamental, qualitatively distinct types of global configurations that a cellular automaton can find itself in. We first define these qualitatively distinct types of dynamical system configurations for the parallel *CA*, and then briefly discuss how these definitions need to be modified in case of *SCA* and *NICA*.

The classification below is based on answering the following question: starting from a given global *CA* configuration, can the automaton return to that same configuration after a finite number of parallel computational steps?

Definition 5 A *fixed point (FP)* is a configuration in the phase space of a *CA* such that, once the *CA* reaches this configuration, it stays there forever. A *cycle configuration (CC)* is a state that, once reached, will be revisited infinitely often with a fixed, finite temporal period of 2 or greater. A *transient configuration (TC)* is a state that, once reached, is never going to be revisited again.

In particular, a FP is a special, degenerate case of a recurrent state with period 1. Due to deterministic evolution, any configuration of a classical, parallel *CA* is either a FP, a proper CC, or a TC. Throughout, we shall make a clear distinction between FPs and “proper” CCs.

On the other hand, if one considers *SCA* so that *arbitrary* node update orderings are permitted, then, given the underlying cellular space and the local update rule, the resulting phase space configurations, due to nondeterminism that results from different choices of possible sequences of node updates (“sequential interleavings”), are more complicated. In a particular *SCA*, a cycle configuration is any configuration revisited infinitely often - but the period between different consecutive visits, assuming an arbitrary sequence s of node updates, need not be fixed. We call a global configuration that is revisited only finitely many times (under a given ordering s) *quasi-cyclic*. Similarly, a *quasi-fixed point* is an *SCA* configuration such that, once the *SCA*’s dynamic evolution reaches this configuration, it stays there “for a while” (i.e., for some finite number of sequential node update steps), and then leaves. For example, a configuration of an *SCA* can simultaneously be both an FP and a quasi-CC, or both a quasi-FP and a CC (see *Subsection 3.1*).

For simplicity, heretofore we shall refer to a configuration C of a *NICA* as a *(weak) fixed point* if there exists some infinite sequence of node updates s such that C is a FP in the usual sense when the automaton’s nodes update according to the ordering s . A *strong fixed point* of a *NICA* automaton is a configuration that is fixed (stable) with respect to *all* possible sequences of node updates. Similarly, we consider a configuration C' to be a cycle state, if there exists an infinite sequence of node updates s' such that, if *NICA* nodes update according to s' , then C' is a cycle state of period 2 or greater in the usual sense (see *Def. 5*). In particular, in case of the *NICA* automata, a single configuration can simultaneously be a weak FP, a CC and a TC; see *Subsection 3.1* for a simple example.

3. 1-D Parallel vs. Sequential CA Comparison and Contrast for Simple Threshold Rules

After the introduction, motivation and the necessary definitions, we now proceed with our main results and their meaning. Technical results (and some of their proofs) are given in this section. Discussion of the implications and relevance of these results, as well as some possible generalizations and extensions, will follow in *Section 4*.

Herein, we compare and contrast the classical, concurrent *CA* with their sequential counterparts, *SCA* and *NICA*, in the context of the simplest nonlinear local update rules possible, viz., the *CA* in which the nodes locally update according to *linear threshold functions*. Moreover, we choose these threshold functions to be *symmetric*, so that the resulting *(S)CA* are also *totalistic* (see, e.g., [8] or [28]). We show the fundamental difference in the configuration spaces, and therefore possible computations, between the parallel threshold automata and the sequential threshold automata: while the former can have temporal cycles (of length two), the computations of the latter always either converge to a fixed point, or otherwise underlying cellular spaces Γ) they fail to finitely converge to any recurrent pattern whatsoever.

For simplicity, but also in order to indicate how dramatically the sequential interleavings of *NICA* fail to capture the concurrency of the classical *CA* based on perfect synchrony, we restrict the underlying cellular spaces to *one-dimensional* Γ . We formally define the class of *1-D (S)CA* of a finite radius below:

Definition 6 A *1-D (sequential) cellular automaton of radius r* ($r \geq 1$) is a *(S)CA* defined over a one-dimensional string of nodes, such that each node's next state depends on the current states of its neighbors to the left and right that are no more than r nodes away. In case of the *(S)CA with memory*, the next state of any node also depends on the current state of that node itself.

Thus, in case of a *Boolean (S)CA with memory* defined over a one-dimensional cellular space Γ , each node's next state depends on exactly $2r + 1$ input bits, while in the *memoryless (S)CA case*, the local update rule is a function of $2r$ input bits. The underlying 1-D cellular space is a string of nodes that can be a finite line graph, a ring (corresponding to the “circular boundary conditions”), a one-way infinite string, or, in the most common case, Γ is a two-way infinite string (or “line”).

We fix the following conventions and terminology. Throughout, only *Boolean CA*, *SCA* and *NICA* are considered; in particular, the set of possible states of any node is $\{0, 1\}$. The phrases “monotone symmetric” and “symmetric (linear) threshold” functions/update rules/automata are used interchangeably. Similarly, “(global) dynamics” and “(global) computation”, when applied to any kind of automata, are used synonymously. Unless stated otherwise, *CA* denotes a classical, concurrent cellular automaton, whereas a cellular automaton where the nodes update sequentially is always denoted by *SCA* (or *NICA*, when appropriate). Also, unless explicitly stated otherwise, *(S)CA with memory* are assumed. The default infinite cellular space Γ is a two-way infinite line. The default finite cellular spaces are finite rings. The terms “phase space” and “configuration space” are used synonymously throughout, as well, and sometimes abridged to *PS* for brevity.

3.1. Synchronous Parallel CA vs. Sequential Interleavings CA: A Simple Example

There are many simple, even trivial examples where not only are concrete computations of the *parallel CA* from particular initial configurations different from the corresponding computations of any of the *sequential CA*, but actually the entire configuration spaces of the parallel *CA* on one, and the corresponding *SCA* and *NICA* on the other hand, turn out to be rather different.

As one of the simplest examples conceivable, consider a trivial *CA* with more than one node (so that talking about “parallel computation” makes sense), namely, a two-node *CA* where each node computes the logical *XOR* of the two inputs. The two phase spaces are given in *Fig. 1*.

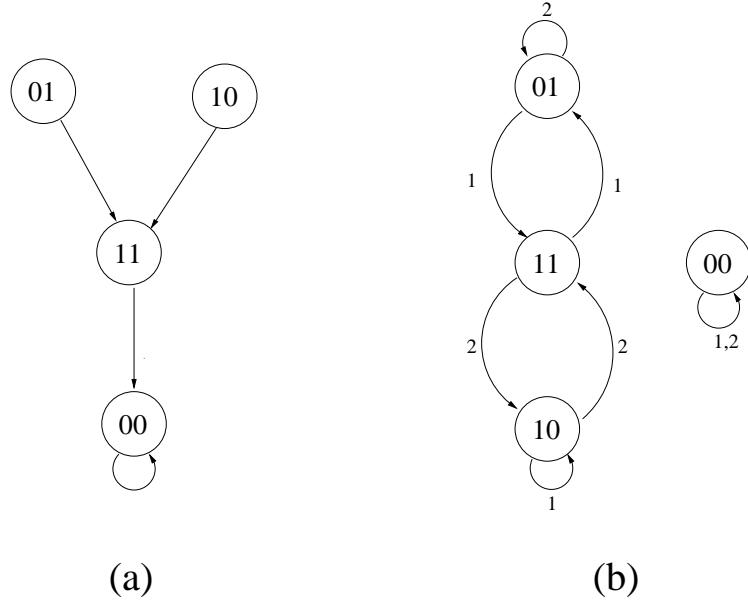


Figure 1:

Configuration spaces for two-node (a) parallel and (b) sequential cellular automata, respectively. Each node computes the logical XOR function of its own current state, and that of the other node. In (b), the integer labels next to the transition arrows indicate which node, 1 or 2, is updating and thus causing the indicated global state transition.

In the parallel case, the state 00 is the “sink”, and the entire configuration space is as in Fig. 1 (a). So, regardless of the starting configuration, after at most two parallel steps, a fixed point “sink” state, that is, in physics terms, a stable global attractor, will be reached.

In the case of sequential node updates, the configuration 00 is still a FP but, this time, it is not reachable from any other configuration. Also, while all three states, 11, 10 and 01, are *transient states* in the parallel case, sequentially, each of them, for any “typical” (infinite) sequence of node updates, is going to be revisited *infinitely often*. In fact, for some sequences of node updates such as, e.g., (1, 1, 2, 2, 2, 1, 2, 2, 1, ...), configurations 01 and 10 are *both quasi-fixed-point states* and *cycle states*. The phase space capturing all possible sequential computations of the two-node automaton with $\delta = \text{XOR}(x_1, x_2)$ for each node is given in Fig. 1 (b). This NICA has three configurations, 01, 10 and 11, each of which is simultaneously a weak FP, a CC and a TC; it is a trivial exercise to find particular update sequences for which each of these configurations is of a desired nature (weak FP, CC or TC). In contrast, configuration 00 is a FP for *any* sequence of node updates^d.

Some observations are in order. First, overall, the configuration space of the *XOR NICA* is richer than the *PS* of its parallel counterpart. In particular, due to determinism, any FP state of a classical *CA* is necessarily a stable attractor or “sink”. In contrast, in case of different possible sequential computations on the same cellular space, the (weak) fixed points clearly need not be stable. Also, whereas the phase space of a parallel *CA* is temporal cycle-free (recall that we do not count FPs among cycles), the phase space of the corresponding *NICA* has nontrivial finite temporal cycles.

^dIn [25] we refer to such FPs of *NICA* as *proper* or *strong* fixed points, in order to contrast them with respect to those configurations that are fixed with respect to *some* but not *all* sequences of the node updates. We also remark that, in a given computation, if the starting configuration of this *NICA*, or any corresponding *SCA*, is different from 00, then this FP configuration is also an example of a *Garden of Eden (GE)* configuration, as it cannot ever be reached irrespective of the sequence s of node updates. For more on *GE* in discrete dynamical systems, the reader is referred to [3, 4].

On the other hand, the union of all possible sequential computations (“interleavings”) cannot fully capture the concurrent computation, either: consider, for example, *reachability* of the state 00.

All these properties can be largely attributed to a relative complexity of the *XOR* function as the update rule, and, in particular, to *XOR*’s *non-monotonicity*. They can also be attributed to the idiosyncrasy of the example chosen. In particular, temporal cycles in the sequential case are not surprising. Also, if one considers *CA* on say four nodes with circular boundary conditions (that is, a *CA* ring on four nodes), these *XOR CA* do have nontrivial cycles in the parallel case, as well. Hence, for *XOR CA* with sufficiently many nodes, the types of computations that the parallel *CA* and the sequential *SCA* and *NICA* are capable of, are quite comparable. Moreover, in those cases where one class is of a richer behavior than the other, it seems reasonable that the *NICA* automata, overall, are capable of more diverse computations than the corresponding synchronous, parallel *CA*, given the nondeterminism of *NICA* arising from all different possibilities for the node update sequences.

This detailed discussion of a trivial example of *CA* and *NICA* phase spaces has the main purpose of motivating what is to follow: an entire class of *CA* and *SCA/NICA*, with the node update functions simpler than *XOR*, yet for which it is the concurrent *CA* that are *provably* capable of a kind of computations that no corresponding (or similar, in the sense to be discussed in *Subsection 3.2* and *Section 4*) *SCA* and, consequently, *NICA*, are capable of.

3.2. On the Existence of Cycles in Threshold Parallel and Sequential Cellular Automata

We shall now compare and contrast the classical, concurrent and perfectly synchronous *CA* with their sequential counterparts, *SCA* and *NICA*, in the context of the simplest nonlinear local update rules possible, namely, the *CA* in which the nodes locally update according to *symmetric linear threshold functions*. This will be done by studying the configuration space properties, that is, the possible computations, of the simple threshold automata in the parallel and sequential settings.

First, we define (*simple*) *linear threshold functions*, and the corresponding types of (*S*)*CA*.

Definition 7 A Boolean-valued linear threshold function of m inputs, x_1, \dots, x_m , is any function of the form

$$f(x_1, \dots, x_m) = \begin{cases} 1, & \text{if } \sum_i w_i \cdot x_i \geq \theta \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

where θ is an appropriate *threshold constant*, and w_1, \dots, w_m are arbitrary (but fixed) real numbers^e called *weights*.

Definition 8 A threshold automaton (*threshold (S)CA*) is a (parallel or sequential) cellular automaton where δ is a Boolean-valued linear threshold function.

Therefore, given an integer k , a *k-threshold function*, in general, is any function of the form as in *Def. 8* with $\theta = k$ and an appropriate choice of weights w_i , $i = 1, \dots, m$. Heretofore we consider *monotonically nondecreasing* Boolean threshold functions only; this, in particular, implies that the weights w_i are always nonnegative. We also additionally assume δ to be a *symmetric function* of all of its inputs. That is, the (*S*)*CA* we analyze have *symmetric, monotone Boolean functions* for their local update rules. We refer to such functions as to *simple threshold functions*, and to the (*S*)*CA* with simple threshold node update rules as to *simple threshold (S)CA*.

^eIn general, w_i can be both positive and negative. This is esp. common in the neural networks literature, where negative weights w_i indicate an *inhibitory effect* of, e.g., one neuron on the firings of another, near-by neuron. In most studies of discrete dynamical systems, however, the weights w_i are required to be nonnegative - that is, only *excitatory effects* of a node on its neighbors are allowed; see, e.g., [3, 4, 26, 27].

Definition 9 A *simple threshold (S)CA* is an automaton whose local update rule δ is a monotone symmetric Boolean (threshold) function.

In particular, if all the weights w_i are positive and equal to one another, then, without loss of generality, we may set them all equal to 1; obviously, this normalization of the weights w_j may also require an appropriate adjustment of the threshold value θ .

Throughout, whenever we say a *threshold automaton* or a *threshold (S)CA*, we shall mean a *simple threshold automaton (threshold (S)CA)*, unless explicitly stated otherwise. That is, the 1-D threshold (S)CA studied in the sequel will have the node update functions of the general form

$$\delta(x_{i-r}, x_{i-r+1}, \dots, x_i, \dots, x_{i+r-1}, x_{i+r}) = \begin{cases} 1, & \text{if } \sum_{j=-r}^r x_{i+j} \geq k \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

where k is a fixed integer from the range $\{0, 1, \dots, 2r + 1, 2r + 2\}$. For example, if the automaton rule radius is $r = 2$, and if $k = 2$, then the k -threshold (S)CA on a specified number of nodes in this case is just the 1-D (S)CA with the node update rule $\delta = \text{"at least 2 out of 5"}$, meaning that the update rule evaluates to 1 if and only if at least two out of five of its inputs are currently equal to 1.

Due to the nature of the node update rules, cyclic behavior intuitively should not be expected in such simple threshold automata. This is, generally, (almost) the case, as will be shown below. We argue that the importance of the results in this subsection largely stems from the following three factors:

- the local update rules are the simplest nonlinear totalistic rules one can think of;
- given the rules, the cycles are not to be expected - yet they exist, and in the case of the classical, parallel (i.e., synchronous) CA *only*; and, related to that observation,
- it is, for this class of automata, the parallel CA that have the more diverse possible dynamics, and, in particular, while qualitatively there is nothing among the possible sequential computations that is not present in the parallel case, the classical parallel threshold CA do exhibit a particular qualitative behavior - they may have nontrivial temporal cycles - that cannot be reproduced by any threshold SCA.

The results below hold for two-way infinite 1-D (S)CA, as well as for finite (S)CA with the circular boundary conditions (i.e., for the (S)CA whose cellular spaces are finite rings).

Lemma 1 *The following dichotomy holds for (S)CA with $\delta = \text{MAJ}$ and $r = 1$:*

(i) *Any 1-D parallel CA with $r = 1$, the MAJORITY update rule, and an even number of nodes, has finite temporal cycles in the phase space (PS); the same holds for two-way infinite 1-D MAJ CA.*

(ii) *1-D Sequential CA with $r = 1$ and the MAJORITY update rule do not have any temporal cycles in the phase space, irrespective of the sequential node update ordering s .*

Remark: In case of the infinite sequential SCA as in the Lemma above, a nontrivial temporal cycle configuration does not exist even in the limit. We also note that s can be an arbitrary sequence of an SCA nodes' indices, not necessarily a (possibly infinitely repeated) permutation, or even a function that is necessarily onto L .

Proof.

To show (i), we exhibit an actual two-cycle. Consider either an infinite 1-D CA, or a finite one, with circular boundary conditions and an even number of nodes, $2n$. Then the configurations $(10)^\omega$ and $(01)^\omega$ in the infinite case ($(10)^n$ and $(01)^n$ in the finite ring case) form a 2-cycle.

To prove (ii), we must show that no cycle is ever possible, irrespective of the starting configuration. We consider all possible 1-neighborhoods (there are eight of them: 000, 001, ..., 111), and show that,

locally, none of them can be cyclic yet not fixed. The case analysis is simple: 000 and 111 are stable (fixed) sub-configurations. Configuration 010, after a single node update, can either stay fixed, or else evolve into any of $\{000, 110, 011\}$; since we are only interested in non-FPs, in the latter case, one can readily show by induction that, after any number of steps, the only additional sub-configuration that can be reached is 111, i.e., assuming that 010 does not stay fixed, $010 \rightarrow^* \{000, 110, 011, 111\}$. However, $010 \notin \{000, 110, 011, 111\}$. By symmetry, similar analysis holds for sub-configuration 101. On the other hand, 110 and 011 either remain fixed, or else at some time step t evolve to 111, which subsequently stays fixed. A similar analysis applies to 001 and 100. Hence, no local neighborhood $x_1x_2x_3$, once it changes, can ever “come back”. Therefore, there are no proper cycles in Sequential 1-D CA with $r = 1$ and $\delta = MAJORITY$. \square

An astute reader may have noticed that the above case analysis in the proof of (ii) can be somewhat simplified if one observes that, for $r = 1$, the sub-configurations 11 and 00 are always stable with respect to the *MAJORITY* node update function, irrespective of the left or right neighbors of the node performing its update, or the updating sequential order.

Part (ii) of *Lemma 1* above can be readily generalized: even if we consider local update rules δ other than the *MAJORITY* rule, yet restrict δ to *monotone symmetric (Boolean) functions* of the input bits, such sequential CA still do not have any proper cycles.

Theorem 1 *For any Monotone Symmetric Boolean 1-D Sequential CA A with $r = 1$, and any sequence s of the node updates, the phase space $PS(A)$ of the automaton A is cycle-free.*

Proof.

Since $r = 1$ and $2r + 1 = 3$, there are only *five* Monotone Symmetric Boolean (*MSB*) functions, or, equivalently, simple threshold functions, on three inputs. Two of these *MSB* functions are utterly trivial (the constant functions 0 and 1). The “at-least-1-out-of-3” simple threshold function is the Boolean *OR* on three inputs; similarly, the “at-least-3-out-of-3” simple threshold function is the Boolean *AND*. It is straight-forward to show that the CA (sequential or parallel, as long as they are *with memory*) with $\delta \in \{\text{OR}, \text{AND}\}$ cannot have temporal cycles. The only remaining *MSB* update rule on three inputs is $\delta = MAJ$, for which we have already argued that the corresponding parallel CA have temporal two-cycles, but all the corresponding *SCA* (and therefore the *NICA*) have cycle-free configuration spaces. \square

Similar results to those in *Lemma 1* and *Theorem 1* also hold for 1-D CA with radius $r = 2$:

Lemma 2 *The following dichotomy holds for (S)CA with $\delta = MAJ$ and $r = 2$:*

- (i) *Many 1-D parallel CA with $r = 2$ and $\delta = MAJ$ have finite cycles in the phase space.*
- (ii) *Any 1-D SCA with $r = 2$ and $\delta = MAJ$, for any sequential order s of the node updates whatsoever, has a cycle-free configuration space.*

Proof.

(i) For $r = 2$, consider configurations $(1100)^\omega$ and $(0011)^\omega$; it is easy to verify that these two configurations form a temporal cycle for the concurrent CA defined over a two-way infinite line.

The argument in (ii) is similar to that of *Lemma 1 (ii)*, except that now there are $2^5 = 32$ fundamental neighborhoods of the form $x_1\dots x_5$ to consider. We notice that, for $r = 2$, the sub-configurations 000 and 111 are stable; this observation simplifies the case analysis. \square

Generalizing Lemmata 1 and 2, part (i), we have the following

Corollary 1 *For all $r \geq 1$, there exists a monotone symmetric CA (that is, a synchronous threshold automaton) A such that A has finite temporal cycles in the phase space.*

Namely, given any $r \geq 1$, a (classical, concurrent) *CA* with $\delta = MAJ$ and $\Gamma = infinite line$ has at least one two-cycle in the *PS*: $\{(0^r 1^r)^\omega, (1^r 0^r)^\omega\}$. If $r \geq 3$ is odd, then such a threshold automaton has at least two distinct two-cycles, since $\{(01)^\omega, (10)^\omega\}$ is also a two-cycle. Analogous results hold for the *threshold CA* defined on finite 1-D cellular spaces, provided that such automata have sufficiently many nodes, that the number of nodes is appropriate (see [25] for more details), and assuming circular boundary conditions (i.e., assuming that Γ is a sufficiently big finite ring). Moreover, the result extends to many finite and infinite *CA* in the higher dimensions, as well; in particular, *threshold CA* with $\delta = MAJ$ that are defined over *2D Cartesian grids* and *Hypervcubes* have two-cycles in their respective phase spaces.

More generally, for any underlying cellular space Γ that is a (finite or infinite) *bipartite graph*, the corresponding (nontrivial) *parallel CA* with $\delta = MAJ$ have temporal two-cycles. We remark that bipartiteness of Γ is sufficient, but it is not necessary, for the existence of temporal two-cycles in this setting.

It turns out that the two-cycles in the *PS* of concurrent *CA* with $\delta = MAJ$ are actually the only type of (proper) temporal cycles such cellular automata can have. Indeed, for any *symmetric linear threshold update rule* δ , and any *finite* regular Cayley graph as the underlying cellular space, the following general result holds [8, 10]:

Proposition 1 *Let a classical, parallel simple threshold CA $A = (\Gamma, N, M)$ be given, where Γ is any finite cellular space, and let this cellular automaton's global map be denoted by F . Then for all configurations $C \in PS(A)$, there exists a finite time step $t \geq 0$ such that $F^{t+2}(C) = F^t(C)$.*

In particular, this result implies that, in case of any *finite* symmetric threshold automaton, for any starting configuration C_0 , there are *only two possible kinds of orbits*: upon repeated iteration, the computation either converges to a fixed point configuration after finitely many steps, or else it eventually arrives at a two-cycle.

It is almost immediate that, if we allow the underlying cellular space Γ to be infinite, if computation from a given starting configuration converges after any finite number of steps at all, it will have to converge either to a fixed point or a two-cycle (but never to a cycle of, say, period three - or any other finite period). The result also extends to finite and infinite *SCA*, provided that we reasonably define what is meant by a single computational step in a situation where the nodes update one at a time. The simplest notion of a single computational step of an *SCA* is that of a single node updating its state. Thus, a single parallel step of a classical *CA* defined on an infinite underlying cellular space Γ includes an infinite amount of sequential computation and, in particular, infinitely many elementary sequential steps. Discussing the implications of this observation, however, is beyond the scope of this work.

Additionally, in order to ensure some sort of convergence of an arbitrary *SCA* (esp. when the underlying Γ is infinite), and, more generally, in order to ensure that *all the nodes* get a chance to update their states, an appropriate condition that guarantees *fairness* needs to be specified. That is, an appropriate restriction on the allowable sequences s of node updates is required. As a first step towards that end, we shall allow only *infinite* sequences s of node updates through the rest of the paper.

For *SCA* defined on finite cellular spaces, one sufficient fairness condition is to impose a fixed upper bound on the number of sequential steps before any given node gets its “turn” to update (again). This is the simplest generalization of the fixed permutation assumption made in the work on sequential and synchronous dynamical systems; see, e.g., [3, 4, 5, 6]. In the infinite *SCA* case, on the other hand, the issue of fairness is nontrivial, and some form of *dove-tailing* of sequential individual node updates may need to be imposed. In the sequel, we shall require from the sequences s of node updates of the *SCA* and *NICA* threshold automata to be fair in a simple sense to be defined shortly, without imposing any further restrictions or investigating how are such fair sequences

of node updates to be generated in a physically realistic distributed setting. For our purposes herein, therefore, the following simple notion of fairness will suffice:

Definition 10 An infinite sequence $s : N \rightarrow L$ is *fair* if (i) the domain L is finite or countably infinite, and (ii) every element $x \in L$ appears infinitely often in the sequence of values $s(1), s(2), s(3), \dots$

Now that we have defined what we mean by a *single step* of a sequential CA, as well as adopted some reasonable notion^f of *fairness*, we can now state the following generalization of *Proposition 1* to both finite and infinite 1-D CA and 1-D SCA:

Proposition 2 Let a parallel CA or a sequential SCA be defined over a finite or infinite 1-D cellular space (that is, a line or a ring), with a finite rule radius $r \geq 1$. Let this automaton's local update rule be an elementary symmetric threshold function. Let's also assume, in the sequential cases, that the fairness condition from Def. 10 holds. Then for any starting configuration $C_0 \in PS(A)$ whatsoever, and any finite subconfiguration $C \subseteq C_0$, there exists a time step $t \geq 0$ such that

$$F^{t+2}(C) = F^t(C) \quad (5)$$

where, in the case of fair SCA, the Eqn. (5) can be replaced with

$$F^{t+1}(C) = F^t(C) \quad (6)$$

In the case of $\delta = MAJ$ (S)CA, a computation starting from any *finitely supported* initial configuration^g necessarily (and quickly [25]) converges to either a FP or a two-cycle [10]:

Proposition 3 Let the assumptions from Proposition 2 hold, and let the underlying threshold rule be $\delta = MAJ$. Then for all configurations $C \in PS(A)$ whatsoever in the finite cases, and for all configurations $C \in PS(A)$ such that C has a finite support when $\Gamma(A)$ is infinite, there exists a finite time step $t \geq 0$ such that $F^{t+2}(C) = F^t(C)$. Moreover, in the sequential cases with fair update sequences, there exists a finite $t \geq 0$ such that $F^{t+1}(C) = F^t(C)$.

Furthermore, if *arbitrary* infinite initial configurations are allowed in *Propositions 2-3*, and the dynamic evolution of the full such global states is monitored, then the only additional possibility is that the particular (S)CA computation fails to finitely converge altogether. In that case, and under the fairness assumption in the case of SCA, the limiting configuration $\lim_{t \rightarrow \infty} F^t(C) = C^{\lim}$ can be shown to be a (stable) fixed point.

To summarize, if the computation of a SCA starting from some configuration C converges at all (that is, to *any* finite temporal cycle), it actually has to converge to a fixed point.

To convince oneself of the validity of *Prop. 2*, two basic facts have to be established. One, convergence to finite temporal cycles of length three or higher is not possible. Indeed, *Prop. 1* establishes that the only possible long-term behaviors of the finite threshold (S)CA are (i) the convergence to a fixed point and (ii) the convergence to a two-cycle. If infinite cellular spaces are considered, it is straight-forward to see that the only new possibility is that the long-term dynamics of a (S)CA fails to (finitely) converge altogether. In some cases with infinite Γ such divergence

^fOur notion of fairness in *Def. 10* need not be the most general, or most suitable in all situations, such a notion. However, it is appropriate for our purposes and, in particular, sufficient for the results on threshold SCA and NICAs that are to follow; see *Prop. 2* in the main text.

^gAlso sometimes called *compact support*; see, e.g., [10]. A global configuration of a cellular automaton defined over an infinite cellular space Γ is said to be *compactly supported* if all except for at most finitely many of the nodes are *quiescent* (i.e., in state 0) in that configuration.

indeed takes place - even when the starting configuration is finitely (compactly) supported: consider, e.g., the *OR* automaton and the starting configuration ...00100... on the two-way infinite line. Two, in the sequential cases (that is, for the simple threshold *SCA* and *NICA*), temporal two-cycles are not possible. That is, a generalization of *Lemmas 1, 2* and *Theorem 1* to arbitrary finite $r \geq 1$, and arbitrary symmetric threshold update rules, holds. This generalization is provided by an appropriate specialization of a similar result in [4] for a class of sequential graph automata called *Sequential Dynamical Systems (SDS)*, with possibly different simple k -threshold update rules at different nodes, and a node update ordering given by repeating *ad infinitum* a (fixed) permutation of the nodes. In particular, part (ii) in the *Theorem 2* below and its proof are directly based on [4]:

Theorem 2 *The following dichotomy holds:*

- (i) *All 1-D (parallel) CA with any odd $r \geq 1$, the local rule $\delta = \text{MAJ}$, and cellular space Γ that is either a finite ring with an even number of nodes or a two-way infinite line, have finite cycles in their phase spaces. The same holds for arbitrary (even or odd) $r \geq 1$ provided that Γ is either a finite ring with a number of nodes divisible by $2r$, or a two-way infinite line^h.*
- (ii) *Any 1-D SCA with any monotone symmetric Boolean update rule δ , any finite $r \geq 1$, defined over any (finite or infinite) 1-D cellular space, and for an arbitrary sequence s (finite or infinite, fair or unfair) as the node update ordering, has a cycle-free phase space.*

Proof.

Part (i): For the special case when $r = 2$, consider the configurations $(1100)^\omega$ and $(0011)^\omega$; it is easy to verify that these two configurations form a cycle for the corresponding parallel *CA*. Similar reasoning readily generalizes to arbitrary $r \geq 2$. The “canonical” temporal two-cycle for *1-D MAJORITY CA* defined over an infinite line with $r \geq 1$ is $\{(1^r 0^r)\}^\omega, (0^r 1^r)^\omega$, with the obvious modification in case of finite *CA* with n nodes (for n even, and assuming circular boundary conditions).

Part (ii) (proof sketch): The proof of this interesting property is based on a slight modification of a similar result in [4] for a class of the sequential graph automata called *Sequential Dynamical Systems (SDS)*. A *simple symmetric SDS* is an *SDS* with (possibly different) k -threshold update rules at different nodes, and with the node update ordering given by a *fixed* permutation of the nodes. The central idea of the proof is to assign nonnegative integer *potentials* to both nodes and edges in the *functional graph* of the given *SCA*. In this functional graph, for any two nodes x_i and x_j , unordered pair $\{x_i, x_j\}$ is an edge if and only if these two nodes provide inputs to one another, i.e., in the 1-D *SCA* case, if and only if $\text{distance}(x_i, x_j) \leq r$ (that is, assuming the canonical labeling of the nodes, so that consecutive nodes always get labeled by consecutive integers, iff $|i - j| \leq r$). The potentials are assigned in such a way that, each time a node changes its value (from 0 to 1 or vice versa), the overall potential of the resulting configuration is strictly less than the overall potential of the configuration before the node flip. Since all individual node and edge potentials are initially nonnegative, and since the total potential of any configuration (that is, the sum of all individual node and edge potentials in this configuration) is always nonnegative, the fact that each “flip” of any node’s value strictly decreases the overall potential by integer amounts implies that, after a finite number of node flips (and, therefore, sequential steps), an equilibrium where no nodes can further flip is reached; this equilibrium will be a fixed point configuration. Due to space considerations, we do not provide all the details. Instead, we again refer the reader to [4] for a full, rigorous proof of the same property as in our claim herein, only in a slightly different setting - the difference being immaterial insofar as the validity of the claim and the applicability of the just outlined proof idea based on the decreasing configuration potentials are concerned.

^hThere are also *CA* defined over finite rings and with even $r \geq 2$ such that the number of nodes in these rings is not divisible by $2r$ yet temporal two-cycles exist. However, a more detailed discussion on what properties the number of nodes in such *CA* has to satisfy is required; we leave this discussion out, however, for the sake of clarity and space constraints.

□

To summarize, symmetric linear threshold CA , depending on the starting configuration, may converge to a fixed point or a temporal two-cycle; in particular, they may end up “looping” in finite (but nontrivial) temporal cycles. In contrast, the corresponding classes of SCA (and therefore $NICA$) can never cycle. We also observe that, given any sequence of node updates of a finite threshold SCA , if this sequence satisfies an appropriate *fairness condition*, then it can be shown that the computation of such a threshold SCA A is guaranteed to converge to a stable fixed-point (sub)configuration on any finite subset of the nodes in $\Gamma(A)$.

The cycle-freeness of the threshold SCA and $NICA$ holds irrespective of the choice of a sequential update ordering (and, extending to infinite SCA , temporal cycles cannot be obtained even “in the limit”ⁱ). Hence, we conclude that no choice of a “sequential interleaving” can capture the perfectly synchronous parallel computation of the parallel threshold CA . Consequently, the “interleaving semantics” of $NICA$ fails to capture the synchronous parallel behavior of the classical CA even for this, simplest nonlinear class of totalistic CA update rules.

4. Discussion and Future Directions

The results in *Section 3* show that, even for the very simplest (nonlinear, nonaffine) totalistic cellular automata, sequential nondeterminism - that is, the union of all possible sequential interleavings - dramatically fails to capture concurrency. It is not surprising that one can find a concurrent CA such that no sequential CA with the same underlying cellular space and the same node update rule can reproduce identical or even “isomorphic” computation, as the example at the beginning of *Section 3* clearly shows (see *Fig. 1* and the related discussion). However, we find it rather interesting that very profound differences can be observed even in the case of extremely simple parallel and sequential CA - that is, the one-dimensional automata with small r and simple threshold functions as the node update rules - and that this profound difference does not apply merely to individual $(S)CA$ and/or their particular computations, but to *all* possible computations of an entire, relatively broad class of the CA update rules.

Moreover, the differences in parallel and sequential computations in the case of the Boolean XOR update rule, for example, can be largely ascribed to the properties of the XOR function (see *Subsection 3.1*). For instance, given that XOR is not *monotone*, the existence of temporal cycles is not at all surprising. In contrast, monotone functions such as $MAJORITY$ are intuitively expected not to have cycles, i.e., for all converging computations, to always converge to a fixed point. This intuition about the monotone symmetric *sequential CA* is shown correct. It is actually, in a sense, “almost correct” for the parallel CA , as well, in that the actual non-FP cycles can be shown to be very few, and without any incoming transients [25]. Thus, in this case, the very existence of the (rare) nontrivial temporal cycles can be ascribed directly to the assumption of *perfect synchrony* of the parallel node updates.

In the actual engineering, physical or biological systems that can be modeled by CA , however, such perfect synchrony is usually hard to justify. In particular, when CA are applied to modeling of various complex physical or biological phenomena (be those the crystal growth, the forest fire propagation, the information or gossip diffusion in a population, or the signal propagation in an organism’s neural system), one ought to primarily focus on the underlying CA behaviors that are, in some sense, *dynamically robust*. This robustness may require, for instance, a *low sensitivity to small perturbations* in the initial configuration. From this standpoint, temporal cycles in the parallel threshold CA are, indeed, an idiosyncrasy of the perfect synchrony, that is, a peculiarity that is anything but robust. Likewise, it makes sense to focus one’s qualitative study of the dynamical

ⁱThat is, via infinitely long computations, obtained by allowing arbitrary infinite sequences of individual node updates.

systems modeled by the threshold CA to those properties that are *statistically robust* (see, e.g., [1]). It can be readily argued in a rigorous, probabilistic sense that, again, the typical, statistically robust behavior of a typical threshold CA computation is a relatively short transient chain, followed by convergence to a stable fixed point. In particular, the non-fixed-point temporal cycles of the threshold CA not only utterly lack any nontrivial basins of attraction (in terms of the incoming transient ‘tails’), but are themselves statistically negligible for all sufficiently large finite, as well as for all infinite CA .

After these digressions on the meaning and implications of our results on the 1-D threshold parallel and sequential *threshold CA*, we now discuss some possible extensions of the results presented thus far. In particular, we are considering extending our study to *non-homogeneous threshold CA*, where not all the nodes necessarily update according to *one and the same* threshold update rule. Another direction of future inquiry is to consider *threshold (S)CA* defined over 2-D and other higher-dimensional regular grids, as well as the *(S)CA* defined over regular Cayley graphs that are not simple Cartesian grids.

One of the more challenging future directions, that have already been explored in other contexts, is to consider CA-like finite automata defined over *arbitrary* (rather than only regular) graphs. Some results on phase space properties of such finite automata with threshold update rules can be found, e.g., in [3, 4]. We have also recently obtained some general results, similar in spirit to those in *Section 3* herein, for the parallel and sequential threshold automata defined over arbitrary *bipartite graphs*: such graph automata, if the nodes are updated concurrently, also in general do contain nontrivial cycles even in case of the simplest node update rules such as *MAJORITY*, yet no cycles are possible if the nodes are updated sequentially and any monotone symmetric node update rule is used.

Another possible extension is to consider classes of the node update rules beyond the simple threshold functions. One obvious candidate are the monotone functions that are not necessarily symmetric (that is, such that the corresponding CA need not be totalistic or semi-totalistic). A possible additional twist, as mentioned above, is to allow for different nodes to update according to different monotone (symmetric or otherwise) local update rules. At what point of the increasing automata complexity, if any, do the possible sequential computations “catch up” with the concurrent ones, appears an interesting problem to consider.

Yet another direction for further investigation is to consider other models of (a)synchrony in cellular automata. We argue that the classical concurrent CA can be viewed, if one is interested in node-to-node interactions among the nodes that are not close to one another, as a class of computational models of *bounded asynchrony*. Namely, if nodes x and y are at distance k (i.e., k nodes apart from each other), and the radius of the CA update rule δ is r , then any change in the state of y can affect the state of x *no sooner*, but also *no later* than after about $\frac{k}{r}$ (parallel node update) computational steps.

We remark that the two particular classes of graph automata defined over arbitrary (not necessarily regular, or Cayley) *finite* graphs, namely, the sequential and synchronous dynamical systems (SDSs and SyDSs, respectively), and their various phase space properties, have been extensively studied; see, e.g., [3, 4, 6, 21] and references therein. It would be interesting, therefore, to consider *asynchronous cellular and graph automata*, where the nodes are not assumed any longer to update in unison and, moreover, where no global clock is assumed. We again emphasize that such automata would entail what can be viewed as *communication asynchrony*, thus going beyond the kind of mere asynchrony in computation at different nodes that has been studied since at least 1984 (e.g., [14, 15]).

What are, then, such *genuinely asynchronous CA* like? How do we specify the local update rules, that is, computations at different nodes, given the possible “communication delays” in what was originally a multiprocessor-like, rather than distributed system-like, parallel model? In the classical, parallel case where a perfect communication synchrony is assumed, any given node x_i of a 1-D CA of radius $r \geq 1$ updates according to

$$x_i^{t+1} = f(x_i^t, x_{i_1}^t, \dots, x_{i_{2r}}^t) \quad (7)$$

for an appropriate local update rule $\delta = f(x_i, x_{i_1}, \dots, x_{i_{2r}})$, whereas, in the asynchronous case, the individual nodes update according to

$$x_i^{t+1} = f(x_i^t, x_{i_1}^{t_1}, \dots, x_{i_{2r}}^{t_{2r}}) \quad (8)$$

We observe that t in *Eqn.* (7) pertains to *the global time*, which of course in this case also coincides with the node x_i 's (and everyone else's) *local time*. However, in case of *Eqn.* (8), each t_j pertains to an appropriate *local time*, in the sense that each $x_{i_j}^{t_j}$ denotes the node x_{i_j} 's value that was most recently received by the node x_i . That is, $x_{i_j}^{t_j}$ is a local view of the node x_{i_j} 's state, as seen by the node x_i . Thus, the nonexistence of the global clock has considerable implications. How to meaningfully relate these different local times, so that one can still mathematically analyze such *ACA* - yet without making the *ACA* description too complicated^j? Yet, if we want to study *genuinely asynchronous CA* models (rather than arbitrary sequential models with global clocks), these changes in the definition seem unavoidable.

We point out that this, genuine (that is, communication) asynchrony in *CA* (see *Eqn.* (8)) can also be readily interpreted in the nondeterministic terms: at each time step, a particular node updates by using its own current value, and also nondeterministically choosing the current or one of the past values of its neighbors. Such a “past value” of a node x_{i_j} used by the node x_i would be only required not to be any “older” than that value of x_{i_j} that x_i had used as its input on its most recent local computation, i.e., on the node x_i 's most recent previous turn to update. That is, from the viewpoint of what are the current inputs to any given node's update function δ , there is a natural nondeterministic interpretation of the fact that the nodes have different clocks.

Many interesting questions arise in this context. One is, what kinds of the phase space properties remain *invariant* under this kind of nondeterminism? Given a triple (Γ, N, M) , it can be readily shown that the fixed points are invariant with respect to the *fair* node update orderings in the (synchronized) sequential *CA*, and, moreover, the FPs are the same for the corresponding parallel *CA*. On the other hand, as our results in *Section 3* indicate, neither cycle configurations nor transient configurations are invariant with respect to whether the nodes are updated sequentially or concurrently (and, in case of the former, in what order). It can be readily observed that, indeed, the (proper, stable) FPs are also invariant for the asynchronous *CA* and graph automata, as well - provided that all the nodes have reached their respective states corresponding to the same fixed point global configuration, and that they all *locally agree* what (sub)configuration they are in, even if their individual local clocks possibly disagree with one another. Therefore, earlier results in [3] on the FP invariance for sequential and synchronous (concurrent) graph automata are just special cases of this, more general result.

Theorem 3 *Given an arbitrary asynchronous cellular or graph automaton, any fixed point configuration is invariant with respect to the choice of a node update ordering, provided that each node x_i has an up-to-date knowledge of the current state of its neighborhood, N_i .*

In addition to studying invariants under different assumptions on asynchrony and concurrency, we also consider qualitative comparison-and-contrast of the asynchronous *CA* that we propose, and the classical *CA*, *SCA* and *NICA*. Such study would shed more light on those behaviors that are solely due to (our abstracted version of) network delays.

^jThat is, while staying away from introducing explicit message *sends* and *receives*, (un)bounded buffers, and the like.

More generally, communication asynchronous *CA*, i.e., the various nondeterministic choices for a given automaton that are due to asynchrony, can be shown to subsume all possible behaviors of the classical and sequential *(S)CA* with the same corresponding (Γ, N, M) . In particular, the nondeterminism that arises from (unbounded) asynchrony subsumes the nondeterminism of a kind studied in *Section 3*; but the question arises, exactly how much more expressive the former model really is than the latter.

5. Summary and Conclusions

We present herein some early steps in studying cellular automata when the unrealistic assumptions of *perfect synchrony* and *instantaneous unbounded parallelism* are dropped. Motivated by the well-known notion of the sequential interleaving semantics of concurrency, we try to apply this metaphor to parallel *CA* and thus motivate the study of sequential cellular automata, *SCA*, and the sequential interleavings automata, *NICA*. In particular, we undertake a comparison and contrast between the *SCA/NICA* and the classical, parallel *CA* models when the node update rules are restricted to *simple threshold functions*. Concretely, we show that, even in very simplistic cases, this sequential “interleaving semantics” of *NICA* fails to capture concurrency of the classical *CA*. One lesson is that, simple as they may be, the basic local operations (i.e., node updates) in the classical *CA* cannot always be considered atomic. That is, the fine-grain parallelism of *CA* turns out not to be quite fine enough for our purposes. It then appears reasonable - indeed, necessary - to consider a single local node update to be made of an ordered sequence of the finer elementary operations:

- Fetching all the neighbors’ values (“Receiving”? “Reading shared variables”?)
- Updating one’s own state according to the update rule δ (that is, performing the local computation)
- Informing the neighbors of the update, i.e., making available one’s new state/value to the neighbors (“Sending”? “Writing a shared variable”?)

Motivated by these early results on the sequential and parallel threshold *CA*, and some of the implications of those results, we next consider various extensions. The central idea is to introduce a class of *genuinely asynchronous CA*, and to formally study their properties. This would hopefully, down the road, lead to some significant insights into the fundamental issues related to bounded vs. unbounded asynchrony, formal sequential semantics for parallel and distributed computation, and, on the cellular automata side, to the identification of many of those classical parallel *CA* phase space properties that are solely or primarily due to the (physically unrealistic) assumption of perfectly synchronous parallel node updates.

We also find various extensions of the basic *CA* model to provide a simple, elegant and useful framework for a high-level study of various global qualitative properties of distributed, parallel and real-time systems at an abstract and rigorous, yet comprehensive level.

Acknowledgments: The work presented herein was supported by the *DARPA IPTO TASK Program*, contract number *F30602-00-2-0586*. Many thanks to Reza Ziae (UIUC) for several useful discussions.

References

1. W. Ross Ashby, "Design for a Brain", Wiley, 1960
2. C. Barrett and C. Reidys, "Elements of a theory of computer simulation I: sequential CA over random graphs", Applied Math. and Computation, vol. 98 (2-3), 1999
3. C. Barrett, H. Hunt, M. Marathe, S. S. Ravi, D. Rosenkrantz, R. Stearns, and P. Tomic, "Gardens of Eden and Fixed Points in Sequential Dynamical Systems", Discrete Math. and Theoretical Comp. Sci. Proc. AA (DM-CCG), July 2001
4. C. Barrett, H. B. Hunt III, M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, R. E. Stearns, "Reachability problems for sequential dynamical systems with threshold functions", TCS vol. 295, issues 1-3, pp. 41-64, Feb. 2003
5. C. Barrett, H. Mortveit, and C. Reidys, "Elements of a theory of computer simulation II: sequential dynamical systems", Applied Math. and Computation, vol. 107 (2-3), 2000
6. C. Barrett, H. Mortveit, and C. Reidys, "Elements of a theory of computer simulation III: equivalence of sequential dynamical systems", Applied Math. and Computation, vol. 122 (3), 2001
7. I. Czaja, R. J. van Glabbek, U. Goltz, "Interleaving semantics and action refinement with atomic choice", in "Advances in Petri Nets" (G. Rozenberg, ed.), LNCS 609, Springer-Verlag, 1992
8. Max Garzon, "Models of Massive Parallelism: Analysis of Cellular Automata and Neural Networks", Springer, 1995
9. R. J. van Glabbek, U. Goltz, "Equivalences and refinement", Proc. LITP Spring School Theoretical CS, La Roche-Posay, France (I. Guessarian, ed.), LNCS 469, Springer-Verlag 1990
10. E. Goles, S. Martinez, "Neural and Automata Networks: Dynamical behavior and Applications", Math. and Its Applications series (vol. 58), Kluwer, 1990
11. E. Goles, S. Martinez (eds.), "Cellular Automata and Complex Systems", Nonlinear Phenomena and Complex Systems series, Kluwer, 1999
12. Howard Gutowitz (ed.), "Cellular Automata: Theory and Experiment", The MIT Press / North-Holland, 1991
13. C. A. R. Hoare, "Communicating Sequential Processes", Prentice Hall, 1985
14. T. E. Ingerson and R. L. Buvel, "Structure in asynchronous cellular automata", Physica D: Nonlinear Phenomena, Volume 10, Issues 1-2, January 1984
15. S. A. Kauffman, "Emergent properties in random complex automata" (ibid.)
16. Robin Milner, "A Calculus of Communicating Systems", Springer-Verlag Lecture Notes in Computer Science (LNCS), 1980
17. Robin Milner, "Calculi for synchrony and asynchrony", Theoretical Computer Sci. 25, pp. 267-310, 1983
18. Robin Milner, "Communication and Concurrency", Prentice-Hall, 1989
19. John von Neumann, "Theory of Self-Reproducing Automata", edited and completed by A. W. Burks, Univ. of Illinois Press, Urbana, 1966
20. J. C. Reynolds, "Theories of Programming Languages", Cambridge Univ. Press, 1998
21. C. Reidys, "On acyclic orientations and sequential dynamical systems", Adv. Appl. Math. vol. 27, 2001
22. Ravi Sethi, "Programming Languages: Concepts & Constructs", 2nd ed., Addison-Wesley, 1996

23. K. Sutner, “Computation theory of cellular automata”, MFCS98 Satellite Workshop on CA, Brno, Czech Rep., 1998
24. P. Tolic, “A Perspective on the Future of Massively Parallel Computing: Fine-Grain vs. Coarse-Grain Parallel Models”, Proc. ACM Computing Frontiers '04, Ischia, Italy, April 2004
25. P. Tolic, G. Agha, “Characterizing Configuration Spaces of Simple Threshold Cellular Automata”, ACRI'04, Amsterdam, The Netherlands, Oct. 25-28, 2004 (to appear in a Springer-Verlag LNCS volume)
26. S. Wolfram “Twenty problems in the theory of CA”, Physica Scripta 9, 1985
27. S. Wolfram (ed.), “Theory and applications of CA”, World Scientific, Singapore, 1986
28. Stephen Wolfram, “Cellular Automata and Complexity (collected papers)”, Addison-Wesley, 1994
29. Stephen Wolfram, “A New Kind of Science”, Wolfram Media, Inc., 2002

Online Efficient Predictive Safety Analysis of Multithreaded Programs

Koushik Sen, Grigore Roșu, Gul Agha

Department of Computer Science,

University of Illinois at Urbana-Champaign.

{ksen,grosu,agha}@cs.uiuc.edu

Abstract. An automated and configurable technique for runtime safety analysis of multithreaded programs is presented, which is able to *predict* safety violations from successful executions. Based on a user provided safety formal specification, the program is automatically instrumented to emit *relevant* state update events to an observer, which further checks them against the safety specification. The events are stamped with *dynamic vector clocks*, enabling the observer to infer a *causal partial order* on the state updates. All event traces that are consistent with this partial order, including the actual execution trace, are analyzed *on-line* and *in parallel*, and a warning is issued whenever there is a trace violating the specification. This technique can be therefore seen as a bridge between testing and model checking. To further increase scalability, a *window* in the state space can be specified, allowing the observer to infer the *most probable* runs. If the size of the window is 1 then only the received execution trace is analyzed, like in testing; if the size of the window is ∞ then all the execution traces are analyzed, such as in model checking.

1 Introduction

In multithreaded systems, threads can execute concurrently communicating with each other through a set of shared variables, yielding an inherent potential for subtle errors due to unexpected interleavings. Both heavy and lighter techniques to detect errors in multithreaded systems have been extensively investigated. The heavy techniques include traditional formal methods based approaches, such as model checking and theorem proving, guaranteeing that a formal model of the system satisfies its safety requirements by exploring, directly or indirectly, all possible thread interleavings. On the other hand, the lighter techniques include testing, that scales well and is one of the most used approaches to validate software products today.

As part of our overall effort in merging testing and formal methods, aiming at getting some of the benefits of both while avoiding the pitfalls of ad hoc testing and the complexity of full-blown model checking or theorem proving, in this paper we present a *runtime verification* technique for safety analysis of multithreaded systems, that can be tuned to analyze from one trace to all traces that are consistent with an actual execution of the program. If all traces are checked, then it becomes equivalent to online model checking of an abstract model of the computation, called the *multithreaded computation lattice*, which is extracted from the actual execution trace of the program, like in POTA [10] or JMPaX [14]. If only one trace is considered, then our technique becomes equivalent to checking just the actual execution of the multithreaded program,

like in testing or like in other runtime analysis tools like MaC [7] and PaX [5, 1]. In general, depending on the application, one can configure a window within the state space to be explored, called *causality cone*, intuitively giving a causal “distance” from the observed execution within which all traces are exhaustively verified. An appealing aspect of our technique is that all these traces can be analyzed *online*, as the events are received from the running program, and all *in parallel* at a cost which in the worst case is proportional with both the size of the window and the size of the state space of the monitor.

There are three important interrelated components of the proposed runtime verification technique namely *instrumentor*, *observer* and *monitor*. The code instrumentor, based on the safety specification, entirely automatically adds code to emit events when *relevant* state updates occur. The observer receives the events from the instrumented program as they are generated, enqueues them and then builds a configurable abstract model of the system, known as a computation lattice, on a layer-by-layer basis. As layers are completed, the monitor, which is synthesized automatically from the safety specification, checks them against the safety specification and then discards them.

The concepts and notions presented in this paper have been experimented and tested on a practical monitoring system for Java programs, JMPaX 2.0, that extends its predecessor JMPaX [12] in at least four non-trivial novel ways. First, it introduces the technical notion of *dynamic vector clock*, allowing it to properly deal with dynamic creation and destruction of threads. Second, the variables shared between threads do not need to be static anymore: an automatic instrumentation technique has been devised that detects automatically when a variable is shared. Thirdly, and perhaps most importantly, the notion of *cone heuristic*, or *global state window*, is introduced for the first time in JMPaX 2.0 to increase the runtime efficiency by analyzing the most likely states in the computation lattice. Lastly, the presented runtime prediction paradigm is safety formalism independent, in the sense that it allows the user to specify any safety property whose bad prefixes can be expressed as a non-deterministic finite automaton (NFA).

2 Monitors for Safety Properties

Safety properties are a very important, if not the most important, class of properties that one should consider in monitoring. This is because once a system violates a safety property, there is no way to continue its execution to satisfy the safety property later. Therefore, a monitor for a safety property can precisely say at runtime when the property has been violated, so that an external recovery action can be taken. From a monitoring perspective, what is needed from a safety formula is a succinct representation of its *bad prefixes*, which are finite sequences of states leading to a violation of the property. Therefore, one can abstract away safety properties by languages over finite words.

Automata are a standard means to succinctly represent languages over finite words. In what follows we define a suitable version of automata, called *monitor*, with the property that it has a “bad” state from which it never gets out:

Definition 1. Let \mathcal{S} be a finite or infinite set, that can be thought of as the set of states of the program to be monitored. Then an \mathcal{S} -monitor or simply a monitor, is a tuple $Mon = \langle \mathcal{M}, m_0, b, \rho \rangle$, where

- \mathcal{M} is the set of states of the monitor;
- $m_0 \in \mathcal{M}$ is the initial state of the monitor;
- $b \in \mathcal{M}$ is the final state of the monitor, also called bad state; and
- $\rho: \mathcal{M} \times \mathcal{S} \rightarrow 2^{\mathcal{M}}$ is a non-deterministic transition relation with the property that $\rho(b, \Sigma) = \{b\}$ for any $\Sigma \in \mathcal{S}$.

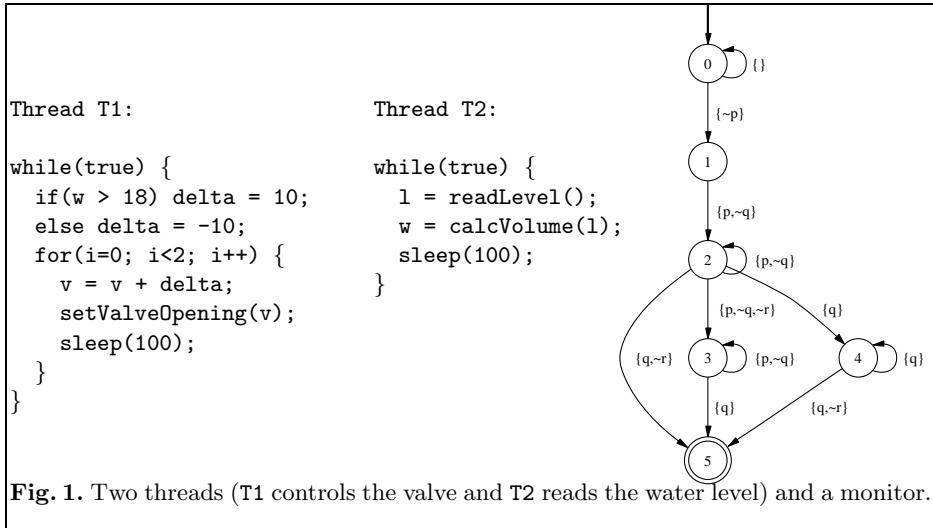
Sequences in \mathcal{S}^* , where ϵ is the empty one, are called (execution) traces. A trace π is said to be a bad prefix in Mon iff $b \in \rho(\{m_0\}, \pi)$, where $\rho: 2^{\mathcal{M}} \times \mathcal{S}^* \rightarrow 2^{\mathcal{M}}$ is recursively defined as $\rho(M, \epsilon) = M$ and $\rho(M, \pi\Sigma) = \rho(\rho(M, \pi), \Sigma)$, where $\rho: 2^{\mathcal{M}} \times \mathcal{S} \rightarrow 2^{\mathcal{M}}$ is defined as $\rho(\{m\} \cup M, \Sigma) = \rho(m, \Sigma) \cup \rho(M, \Sigma)$ and $\rho(\emptyset, \Sigma) = \emptyset$, for all finite $M \subseteq \mathcal{M}$ and $\Sigma \in \mathcal{S}$.

\mathcal{M} is not required to be finite in the above definition, but $2^{\mathcal{M}}$ represents the set of *finite* subsets of \mathcal{M} . In practical situations it is often the case that the monitor is *not* explicitly provided in a mathematical form as above. For example, a monitor can be just any program whose execution is triggered by receiving events from the monitored program; its state can be given by the values of its local variables, and the bad state has some easy to detect property, such as a specific variable having a negative value.

There are fortunate situations in which monitors can be *automatically generated* from formal specifications, thus requiring the user to focus on system's formal safety requirements rather than on low level implementation details. In fact, this was the case in all the experiments that we have performed so far. We have so far experimented with requirements expressed either in extended regular expressions (ERE) or various variants of temporal logics, with both future and past time. For example, [11, 13] show coinductive techniques to generate minimal static monitors from EREs and from future time linear temporal logics, respectively, and [6, 1] show how to generate dynamic monitors, i.e., monitors that generate their states on-the-fly, as they receive the events, for the safety segment of temporal logic.

Example 1. Consider a reactive controller that maintains the water level of a reservoir within safe bounds. It consists of a water level reader and a valve controller. The water level reader reads the current level of the water, calculates the quantity of water in the reservoir and stores it in a shared variable w . The valve controller controls the opening of a valve by looking at the current quantity of water in the reservoir. A very simple and naive implementation of this system contains two threads: T1, the valve controller, and T2, the water level reader. The code snippet for the implementation is given in Fig. 1. Here w is in some proper units such as mega gallons and v is in percentage. The implementation is poorly synchronized and it relies on ideal thread scheduling.

A sample run of the system can be $\{w = 20, v = 40\}, \{w = 24\}, \{v = 50\}, \{w = 27\}, \{v = 60\}, \{w = 31\}, \{v = 70\}$. As we will see later in the paper, by a run we here mean a sequence of relevant variable writes. Suppose we are



interested in a safety property that says “If the water quantity is more than 30 mega gallons, then it is the case that sometime in the past water quantity exceeded 26 mega gallons and since then the valve is open by more than 55% and the water quantity never went down below 26 mega gallon”. We can express this safety property in two different formalisms: linear temporal logic (LTL) with both past-time and future-time, or extended regular expressions (EREs) for bad prefixes. The atomic propositions that we will consider are $p : (w > 26)$, $q : (w > 30)$, $r : (v > 55)$. The properties can be written as follows:

$$F_1 = \square(q \rightarrow ((r \wedge p)\mathcal{S} \uparrow p)) \quad (1)$$

$$F_2 = \{\}^* \{\neg p\} \{p, \neg q\}^+ (\{p, \neg q, \neg r\} \{p, \neg q\}^* \{q\} + \{q\}^* \{q, \neg r\}) \{\}^* \quad (2)$$

The formula F_1 in LTL ($\uparrow p$ is a shorthand for “ p and previously not p ”) states that “It is always the case that if $(w > 30)$ then at some time in the past $(w > 26)$ started to be true and since then $(r > 55)$ and $(w > 26)$.” The formula F_2 characterizes the prefixes that make F_1 false. In F_2 we use $\{p, \neg q\}$ to denote a state where p and $\neg q$ holds and r may or may not hold. Similarly, $\{\}$ represents any state of the system. The monitor automaton for F_2 is given also in Fig. 1.

3 Multithreaded Programs

We consider multithreaded systems in which threads communicate with each other via shared variables. A crucial point is that some variable updates can causally depend on others. We will describe an efficient *dynamic vector clock* algorithm which, given an executing multithreaded program, generates appropriate messages to be sent to an external observer. Section 4 will show how the observer, in order to perform its more elaborated analysis, extracts the state update information from such messages together with the causality partial order.

3.1 Multithreaded Executions and Shared Variables

A multithreaded program consists of n threads t_1, t_2, \dots, t_n that execute concurrently and communicate with each other through a set of shared variables. A

multithreaded execution is a sequence of events $e_1 e_2 \dots e_r$ generated by the running multithreaded program, each belonging to one of the n threads and having type *internal*, *read* or *write* of a shared variable. We use e_i^j to represent the j -th event generated by thread t_i since the start of its execution. When the thread or position of an event is not important we can refer to it generically, such as e , e' , etc.; we may write $e \in t_i$ when event e is generated by thread t_i . Let us fix an arbitrary but fixed multithreaded execution, say \mathcal{M} , and let S be the set of all variables that were shared by more than one thread in the execution. There is an immediate notion of *variable access precedence* for each shared variable $x \in S$: we say e *x-precedes* e' , written $e <_x e'$, iff e and e' are variable access events (reads or writes) to the same variable x , and e “happens before” e' , that is, e occurs before e' in \mathcal{M} . This can be realized in practice by keeping a counter for each shared variable, which is incremented at each variable access.

3.2 Causality and Multithreaded Computations

Let \mathcal{E} be the set of events occurring in \mathcal{M} and let \prec be the partial order on \mathcal{E} :

- $e_i^k \prec e_i^l$ if $k < l$;
- $e \prec e'$ if there is $x \in S$ with $e <_x e'$ and at least one of e , e' is a write;
- $e \prec e''$ if $e \prec e'$ and $e' \prec e''$.

We write $e||e'$ if $e \not\prec e'$ and $e' \not\prec e$. The tuple (\mathcal{E}, \prec) is called the *multithreaded computation* associated with the original multithreaded execution \mathcal{M} . Synchronization of threads can be easily and elegantly taken into consideration by just generating dummy read/write events when synchronization objects are acquired/released, so the simple notion of multithreaded computation as defined above is as general as practically needed. A permutation of all events e_1, e_2, \dots, e_r that does not violate the multithreaded computation, in the sense that the order of events in the permutation is consistent with \prec , is called a *consistent multithreaded run*, or simply, a *multithreaded run*.

A multithreaded computation can be thought of as the *most general assumption* that an observer of the multithreaded execution can make about the system without knowing what it is supposed to do. Indeed, an external observer simply *cannot disregard* the order in which the same variable is modified and used within the observed execution, because this order can be part of the intrinsic semantics of the multithreaded program. However, multiple consecutive reads of the same variable can be permuted, and the particular order observed in the given execution is not critical. As seen in Section 4, by allowing an observer to analyze *multithreaded computations* rather than just *multithreaded executions*, one gets the benefit of not only properly dealing with potential re-orderings of delivered messages (e.g., due to using multiple channels in order to reduce the monitoring overhead), but especially of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling.

3.3 Relevant Causality

Some of the variables in S may be of no importance at all for an external observer. For example, consider an observer whose purpose is to check the property “if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was

always false”; formally, using the interval temporal logic notation in [6], this can be compactly written as $(x > 0) \rightarrow [y = 0, y > z]$. All the other variables in S except x , y and z are essentially irrelevant for this observer. To minimize the number of messages, like in [8] which suggests a similar technique but for distributed systems in which reads and writes are not distinguished, we consider a subset $\mathcal{R} \subseteq \mathcal{E}$ of *relevant events* and define the \mathcal{R} -*relevant causality* on \mathcal{E} as the relation $\triangleleft := \prec \cap (\mathcal{R} \times \mathcal{R})$, so that $e \triangleleft e'$ iff $e, e' \in \mathcal{R}$ and $e \prec e'$. It is important to notice though that the other variables can also indirectly influence the relation \triangleleft , because they can influence the relation \prec . We next provide a technique based on *vector clocks* that correctly implements the relevant causality relation.

3.4 Dynamic Vector Clock Algorithm

We provide a technique based on *vector clocks* [4, 9] that correctly and efficiently implements the relevant causality relation. Let $V : ThreadId \rightarrow Nat$ be a *partial* map from thread identifiers to natural numbers. We call such a map a *dynamic vector clock (DVC)* because its partiality reflects the intuition that threads are dynamically created and destroyed. To simplify the exposition and the implementation, we assume that each DVC V is a total map, where $V[t] = 0$ whenever V is not defined on thread t .

We associate a DVC with every thread t_i and denote it by V_i . Moreover, we associate two DVCs V_x^a and V_x^w with every shared variable x ; we call the former *access DVC* and the latter *write DVC*. All the DVCs V_i are kept empty at the beginning of the computation, so they do not consume any space. For DVCs V and V' , we say that $V \leq V'$ if and only if $V[j] \leq V'[j]$ for all j , and we say that $V < V'$ iff $V \leq V'$ and there is some j such that $V[j] < V'[j]$; also, $\max\{V, V'\}$ is the DVC with $\max\{V, V'\}[j] = \max\{V[j], V'[j]\}$ for each j . Whenever a thread t_i with current DVC V_i processes event e_i^k , the following algorithm is executed:

1. if e_i^k is relevant, i.e., if $e_i^k \in \mathcal{R}$, then

$$V_i[i] \leftarrow V_i[i] + 1$$
2. if e_i^k is a read of a variable x then

$$V_i \leftarrow \max\{V_i, V_x^w\}$$

$$V_x^a \leftarrow \max\{V_x^a, V_i\}$$
3. if e_i^k is a write of a variable x then

$$V_x^w \leftarrow V_x^a \leftarrow V_i \leftarrow \max\{V_x^a, V_i\}$$
4. if e_i^k is relevant then
send message $\langle e_i^k, i, V_i \rangle$ to observer.

The following theorem states that the DVC algorithm correctly implements causality in multithreaded programs. This algorithm has been previously presented by the authors in [14, 15] in a less general context, where the number of threads is fixed and known a priori. Its proof is similar to that in [15].

Theorem 1. *After event e_i^k is processed by thread t_i ,*

- $V_i[j]$ equals the number of relevant events of t_j that causally precede e_i^k ; if $j = i$ and e_i^k is relevant then this number also includes e_i^k ;
- $V_x^a[j]$ equals the number of relevant events of t_j that causally precede the most recent event that accessed (read or wrote) x ; if $i = j$ and e_i^k is a relevant read or write of x event then this number also includes e_i^k ;

- $V_x^w[j]$ equals the number of relevant events of t_j that causally precede the most recent write event of x ; if $i = j$ and e_i^k is a relevant write of x then this number also includes e_i^k .

Therefore, if $\langle e, i, V \rangle$ and $\langle e', j, V' \rangle$ are two messages sent by dynamic vector clock algorithm, then $e \triangleleft e'$ if and only if $V[i] \leq V'[i]$. Moreover, if i and j are not given, then $e \triangleleft e'$ if and only if $V < V'$.

4 Runtime Model Generation and Predictive Analysis

In this section we consider what happens at the observer’s site. The observer receives messages of the form $\langle e, i, V \rangle$. Because of Theorem 1, the observer can infer the causal dependency between the relevant events emitted by the multithreaded system. We show how the observer can be configured to effectively analyze all possible interleavings of events that do not violate the observed causal dependency *online* and *in parallel*. Only one of these interleavings corresponds to the real execution, the others being all potential executions. Hence, the presented technique can *predict* safety violations from successful executions.

4.1 Multithreaded Computation Lattice

Inspired by related definitions in [2], we define the important notions of relevant multithreaded computation and run as follows. A *relevant multithreaded computation*, simply called *multithreaded computation* from now on, is the partial order on events that the observer can infer, which is nothing but the relation \triangleleft . A *relevant multithreaded run*, also simply called *multithreaded run* from now on, is any permutation of the received events which *does not violate* the multithreaded computation. Our major purpose in this paper is to check safety requirements against *all* (relevant) multithreaded runs of a multithreaded system.

We assume that the relevant events are only writes of shared variables that appear in the safety formulae to be monitored, and that these events contain a pair of the name of the corresponding variable and the value which was written to it. We call these variables *relevant variables*. Note that events can change the state of the multithreaded system as seen by the observer; this is formalized next. A *relevant program state*, or simply a *program state* is a map from relevant variables to concrete values. Any permutation of events generates a sequence of program states in the obvious way, however, not all permutations of events are valid multithreaded runs. A program state is called *consistent* if and only if there is a multithreaded run containing that state in its sequence of generated program states. We next formalize these concepts.

We let \mathcal{R} denote the set of received relevant events. For a given permutation of events in \mathcal{R} , say $e_1 e_2 \dots e_{|\mathcal{R}|}$, we let e_i^k denote the k -th event of thread t_i . Then the relevant program state after the events $e_1^{k_1}, e_2^{k_2}, \dots, e_n^{k_n}$ is called a *relevant global multithreaded state*, or simply a *relevant global state* or even just *state*, and is denoted by $\Sigma^{k_1 k_2 \dots k_n}$. A state $\Sigma^{k_1 k_2 \dots k_n}$ is called *consistent* if and only if for any $1 \leq i \leq n$ and any $l_i \leq k_i$, it is the case that $l_j \leq k_j$ for any $1 \leq j \leq n$ and any l_j such that $e_j^{l_j} \triangleleft e_i^{l_i}$. Let Σ^{K_0} be the *initial global state*, $\Sigma^{00\dots 0}$. An important observation is that $e_1 e_2 \dots e_{|\mathcal{R}|}$ is a multithreaded run if and only if

it generates a sequence of global states $\Sigma^{K_0} \Sigma^{K_1} \dots \Sigma^{K_{|\mathcal{R}|}}$ such that each Σ^{K_r} is consistent and for any two consecutive Σ^{K_r} and $\Sigma^{K_{r+1}}$, K_r and K_{r+1} differ in exactly one index, say i , where the i -th element in K_{r+1} is larger by 1 than the i -th element in K_r . For that reason, we will identify the sequences of states $\Sigma^{K_0} \Sigma^{K_1} \dots \Sigma^{K_{|\mathcal{R}|}}$ as above with multithreaded runs, and simply call them *runs*.

We say that Σ *leads-to* Σ' , written $\Sigma \rightsquigarrow \Sigma'$, when there is some run in which Σ and Σ' are consecutive states. Let \rightsquigarrow^* be the reflexive transitive closure of the relation \rightsquigarrow . The set of all consistent global states together with the relation \rightsquigarrow^* forms a *lattice* with n mutually orthogonal axes representing each thread. For a state $\Sigma^{k_1 k_2 \dots k_n}$, we call $k_1 + k_2 + \dots + k_n$ its *level*. A *path* in the lattice is a sequence of consistent global states on increasing level, where the level increases by 1 between any two consecutive states in the path. Therefore, a run is just a path starting with $\Sigma^{00\dots 0}$ and ending with $\Sigma^{r_1 r_2 \dots r_n}$, where r_i is the total number of events of thread t_i . Note that in the above discussion we assumed a fixed number of threads n . In a program where threads can be created and destroyed dynamically, only those threads are considered that at the end of the computation have causally affected the final values of the relevant variables.

Therefore, a multithreaded computation can be seen as a lattice. This lattice, which is called *computation lattice* and referred to as \mathcal{L} , should be seen as an *abstract model* of the running multithreaded program, containing the relevant information needed in order to analyze the program. Supposing that one is able to *store* the computation lattice of a multithreaded program, which is a non-trivial matter because it can have an exponential number of states in the length of the execution, one can mechanically model-check it against the safety property. *Example 2.* Figure 2 shows the causal partial order on relevant events extracted by the observer from the multithreaded execution in Example 1, together with the generated computation lattice. The actual execution, $\Sigma^{00} \Sigma^{01} \Sigma^{11} \Sigma^{12} \Sigma^{22} \Sigma^{23} \Sigma^{33}$, is marked with solid edges in the lattice. Besides its DVC, each global state in the lattice stores its values for the relevant variables, w and v . It can be readily seen on Fig. 2 that the LTL property F_1 defined in Example 1 holds on the sample run of the system, and also that it is not in the language of bad prefixes, F_2 . However, F_1 is violated on some other consistent runs, such as $\Sigma^{00} \Sigma^{01} \Sigma^{02} \Sigma^{12} \Sigma^{13} \Sigma^{23} \Sigma^{33}$. On this particular run $\uparrow p$ holds at Σ^{02} ; however, r does not hold at the next state Σ^{12} . This makes the formula F_1 false at the state Σ^{13} . The run can also be symbolically written as $\{\} \{\} \{p\} \{p\} \{p, q\} \{p, q, r\} \{p, q, r\}$. In the automaton in Fig. 1, this corresponds to a possible sequence of states 00123555. Hence, this string is accepted by F_2 as a bad prefix.

Therefore, by carefully analyzing the computation lattice extracted from a successful execution one can infer safety violations in other possible consistent executions. Such violations give informative feedback to users, such as the lack of synchronization in the example above, and may be hard to find by just ordinary testing. In what follows we propose effective techniques to analyze the computation lattice. A first important observation is that one can generate it *on-the-fly* and analyze it on a level-by-level basis, discarding the previous levels. However,

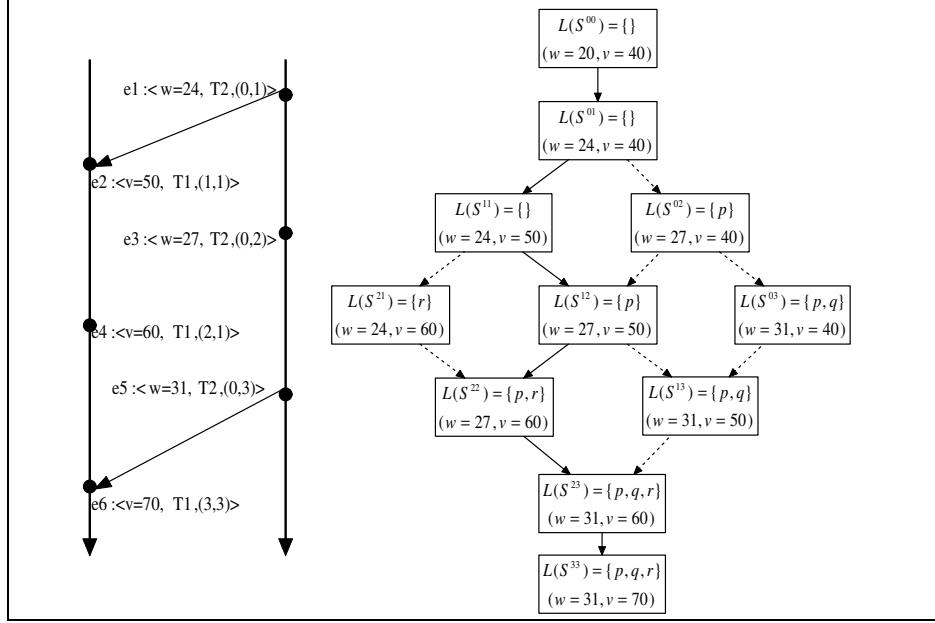


Fig. 2. Computation Lattice

even if one considers only one level, that can still contain an exponential number of states in the length of the current execution. A second important observation is that the states in the computation lattice are not all equiprobable in practice. By allowing a user configurable *window* of most likely states in the lattice centered around the observed execution trace, the presented technique becomes quite scalable, requiring $O(wm)$ space and $O(twm)$ time, where w is the size of the window, m is the size of the bad prefix monitor of the safety property, and t is the size of the monitored execution trace.

4.2 Level By Level Analysis of the Computation Lattice

A naive observer of an execution trace of a multithreaded program would just check the observed execution trace against the monitor for the safety property, say $\mathcal{M}on$ like in Definition 1, and would maintain at each moment a set of states, say $\mathcal{M}onStates$ in \mathcal{M} . When a new event generating a new global state Σ arrives, it would replace $\mathcal{M}onStates$ by $\rho(\mathcal{M}onStates, \Sigma)$. If the bad state b will ever be in $\mathcal{M}onStates$ then a property violation error would be reported, meaning that the current execution trace led to a bad prefix of the safety property. Here we assume that the events are received in the order in which they are emitted, and also that the monitor works over the global states of the multithreaded programs.

A smart observer, as said before, will analyze not only the observed execution trace, but also all the other consistent runs of the multithreaded system, thus being able to *predict* violations from successful executions. The observer receives the events from the running multithreaded program in real-time and enqueues

them in an event queue Q . At the same time, it traverses the computation lattice level by level and checks whether the bad state of the monitor can be hit by any of the runs up to the current level. We next provide the algorithm that the observer uses to construct the lattice level by level from the sequence of events it receives from the running program.

The observer maintains a list of global states ($CurrLevel$), that are present in the current level of the lattice. For each event e in the event queue, it tries to construct a new global state from the set of states in the current level and the event e . If the global state is created successfully then it is added to the list of global states ($NextLevel$) for the next level of the lattice. The process continues until certain condition, $levelComplete?()$ holds. At that time the observer says that the level is complete and starts constructing the next level by setting $CurrLevel$ to $NextLevel$ and reallocating the space previously occupied by $CurrLevel$. Here the predicate $levelComplete?()$ is crucial for generating only those states in the level that are most likely to occur in other executions, namely those in the *window*, or the *causality cone*, that is described in the next subsection. The $levelComplete?$ predicate is also discussed and defined in the next subsection. The pseudo-code for the lattice traversal is given in Fig. 3.

Every global state Σ contains the value of all relevant shared variables in the program, a DVC $VC(\Sigma)$ to represent the latest events from each thread that resulted in that global state. Here the predicate $nextState?(\Sigma, e)$, checks if the event e can convert the state Σ to a state Σ' in the next level of the lattice, where $threadId(e)$ returns the index of the thread that generated the event e , $VC(\Sigma)$ returns the DVC of the global state Σ , and $VC(e)$ returns the DVC of the event e . It essentially says that event e can generate a consecutive state for a state Σ , if and only if Σ “knows” everything e knows about the current evolution of the multithreaded system except for the event e itself. Note that e may know less than Σ knows with respect to the evolution of other threads in the system, because Σ has global information.

The function $createState(\Sigma, e)$ creates a new global state Σ' , where Σ' is a possible consistent global state that can result from Σ after the event e . Together with each state Σ in the lattice, a set of states of the monitor, $MonStates(\Sigma)$, also needs to be maintained, which keeps all the states of the monitor in which any of the partial runs ending in Σ can lead to. In the function $createState$, we set the $MonStates$ of Σ' with the set of monitor states to which any of the current states in $MonStates(\Sigma)$ can transit within the monitor when the state Σ' is observed. $pgmState(\Sigma')$ returns the value of all relevant program shared variables in state Σ' , $var(e)$ returns the name of the relevant variable that is written at the time of event e , $value(e)$ is the value that is written to $var(e)$, and $pgmState(\Sigma')[var(e) \leftarrow value(e)]$ means that in $pgmState(\Sigma')$, $var(e)$ is updated with $value(e)$.

The merging operation $nextLevel \uplus \Sigma$ adds the global state Σ to the set $nextLevel$. If Σ is already present in $nextLevel$, it updates the existing state’s $MonStates$ with the union of the existing state’s $MonStates$ and the $Monstates$ of Σ . Two global states are same if their DVCs are equal. Because of the function

levelComplete?, it may be often the case that the analysis procedure moves from the current level to the next one before it is exhaustively explored. That means that several events in the queue, which were waiting for other events to arrive in order to generate new states in the current level, become unnecessary so they can be discarded. The function *removeUselessEvents(CurrLevel, Q)* removes from Q all the events that cannot contribute to the construction of any state at the next level. To do so, it creates a DVC V_{min} whose each component is the minimum of the corresponding component of the DVCs of all the global states in the set *CurrLevel*. It then removes all the events in Q whose DVCs are less than or equal to V_{min} . This function makes sure that we do not store any unnecessary events.

```

while(not end of computation){
     $Q \leftarrow enqueue(Q, NextEvent())$ 
    while(constructLevel()){}
}

boolean constructLevel{
    for each  $e \in Q$  {
        if  $\Sigma \in CurrLevel$  and nextState?( $\Sigma, e$ ) {
             $NextLevel \leftarrow NextLevel \uplus createState(\Sigma, e)$ 
            if levelComplete?( $NextLevel, e, Q$ ) {
                 $Q \leftarrow removeUselessEvents(CurrLevel, Q)$ 
                 $CurrLevel \leftarrow NextLevel$ 
                return true}
        }
        return false
    }
    boolean nextState?( $\Sigma, e$ ){
         $i \leftarrow threadId(e)$ ;
        if ( $\forall j \neq i : VC(\Sigma)[j] \geq VC(e)[j]$  and
             $VC(\Sigma)[i] + 1 = VC(e)[i]$ ) return true
        return false
    }
    State createState( $\Sigma, e$ ){
         $\Sigma' \leftarrow \text{new copy of } \Sigma$ 
         $j \leftarrow threadId(e); VC(\Sigma')[j] \leftarrow VC(\Sigma)[j] + 1$ 
         $pgmState(\Sigma')[var(e)] \leftarrow value(e)$ 
         $MonStates(\Sigma') \leftarrow \rho(MonStates(\Sigma), \Sigma')$ 
        if  $b \in MonStates(\Sigma')$  {
            output 'property may be violated'
        }
        return  $\Sigma'$ 
    }

```

Fig. 3. Level-by-level traversal.

The observer runs in a loop till the computation ends. In the loop the observer waits for the next event from the running instrumented program and enqueues it in Q whenever it becomes available. After that the observer runs the function *constructLevel* in a loop till it returns false. If the function *constructLevel* returns false then the observer knows that the level is not completed and it needs more events to complete the level. At that point the observer again starts waiting for the next event from the running program and continues with the loop. The pseudo-code for the observer is given at the top of Fig. 3.

4.3 Causality Cone Heuristic

In a given level of a computation lattice, the number of states can be large; in fact, exponential in the length of the trace. In online analysis, generating all the states in a level may not be feasible. However, note that some states in a level can be considered more likely to occur in a consistent run than others. For example, two independent events

that can possibly permute may have a huge time difference. Permuting these two events would give a consistent run, but that run may not be likely to take place in a real execution of the multithreaded program. So we can ignore such a

permutation. We formalize this concept as *causality cone*, or *window*, and exploit it in restricting our attention to a small set of states in a given level.

In what follows we assume that the events are received in an order in which they happen in the computation. This is easily ensured by proper instrumentation. Note that this ordering gives the real execution of the program and it respects the partial order associated with the computation. This execution will be taken as a reference in order to compute the most probable consistent runs of the system.

If we consider all the events generated by the executing distributed program as a finite sequence of events, then a lattice formed by any prefix of this sequence is a sublattice of the computation lattice \mathcal{L} . This sublattice, say \mathcal{L}' has the following property: if $\Sigma \in \mathcal{L}'$, then for any $\Sigma' \in \mathcal{L}$ if $\Sigma' \rightsquigarrow^* \Sigma$ then $\Sigma' \in \mathcal{L}'$. We can see this sublattice as a portion of the computation lattice \mathcal{L} enclosed by a cone. The height of this cone is determined by the length of the current sequence of events. We call this *causality cone*. All the states in \mathcal{L} that are outside this cone cannot be determined from the current sequence of events. Hence, they are outside the causal scope of the current sequence of events. As we get more events this cone moves down by one level.

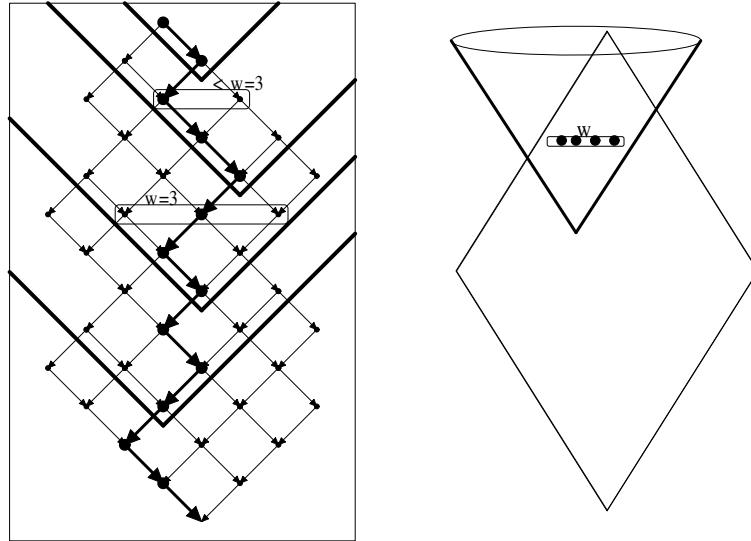


Fig. 4. Causality Cones

If we compute a DVC V_{max} whose each component is the maximum of the corresponding component of the DVCs of all the events in the event queue, then this represents the DVC of the global state appearing at the tip of the cone. The tip of the cone traverses the actual execution run of the program.

To avoid the generation of possibly exponential number of states in a given level, we consider a fixed number, say w , most probable states in a given level. In

a level construction we say the level is complete once we have generated w states in that level. However, a level may contain less than w states. Then the level construction algorithm gets stuck. Moreover, we cannot determine if a level has less than w states unless we see all the events in the complete computation. This is because we do not know the total number of threads that participate in the computation beforehand. To avoid this scenario we introduce another parameter l , the length of the current event queue. We say that a level is complete if we have used all the events in the event queue for the construction of the states in the current level and the length of the queue is l and we have not crossed the limit w on the number of states. The pseudo-code for *levelComplete?* is given in Fig. 5

Note, here l corresponds to the number of levels of the sublattice that be constructed from the events in the event queue Q . On the other hand, the level of this sublattice with the largest level number and having at least w global states refers to the *CurrLevel* in the algorithm.

```
boolean levelComplete?(NextLevel, e, Q){
    if size(NextLevel)  $\geq w$  then
        return true;
    else if e is the last event in Q
        and size(Q) == l then
            return true;
        else return false;
}
```

Fig. 5. *levelComplete?* predicate

5 Implementation

We have implemented these new techniques, in version 2.0 of the tool Java MultiPathExplorer (JMPaX)[12], which has been designed to monitor multithreaded Java programs. The current implementation is written in Java and it removes the restriction that all the shared variables of the multithreaded program are static variables of type **int**. The tool has three main modules, the *instrumentation* module, the *observer* module and the *monitor* module.

The instrumentation program, named **instrument**, takes a specification file and a list of class files as command line arguments. An example is

```
java instrument spec A.class B.class C.class
```

where the specification file **spec** contains a list of named formulae written in a suitable logic. The program **instrument** extracts the name of the relevant variables from the specification and instruments the classes, provided in the argument, as follows:

- i) For each variable x of primitive type in each class it adds *access* and *write* DVCs, namely `_access_dvc_x` and `_write_dvc_x`, as new fields in the class.
- ii) It adds code to associate a DVC with every newly created thread;
- iii) For each read and write access of a variable of primitive type in any class, it adds codes to update the DVCs according to the algorithm mentioned in Section 3.4;
- iv) It adds code to call a method `handleEvent` of the *observer* module at every write of a relevant variable.

The instrumentation module uses BCEL [3] Java library to modify Java class files. We use the BCEL library to get a better handle for a Java classfile.

The *observer* module, that takes two parameters w and l , generates the lattice level by level when the instrumented program is executed. Whenever the `handleEvent` method is invoked it enqueues the event passed as argument to the method `handleEvent`. Based on the event queue and the current level of the lattice it generates the next level of the lattice. In the process it invokes `nextStates` method (corresponding to ρ in a *monitor*) of the *monitor* module.

The *monitor* module reads the specification file written either as an LTL formula or a regular expression and generates the non-deterministic automaton corresponding to the formula or the regular expression. It provides the method `nextStates` as an interface to the *observer* module. The method raises an exception if at any point the set of states returned by `nextStates` contain the “bad” state of the automaton. The system being modular, user can plug in his/her own *monitor* module for his/her logic of choice.

Since in Java synchronized blocks cannot be interleaved, so corresponding events cannot be permuted, locks are considered as shared variables and a write event is generated whenever a lock is acquired or released. This way, a causal dependency is generated between any exit and any entry of a synchronized block, namely the expected happens-before relation. Java synchronization statements are handled exactly the same way, that is, the shared variable associated to the synchronization object is written at the entrance and at the exit of the synchronized region. Condition synchronizations (wait/notify) can be handled similarly, by generating a write of a dummy shared variable by both the notifying thread before notification and by the notified thread after notification.

6 Conclusion and Future Work

A formal runtime predictive analysis technique for multithreaded systems has been presented in this paper, in which multiple threads communicating by shared variables are automatically instrumented to send relevant events, stamped by dynamic vector clocks, to an external observer which extracts a causal partial order on the global state, updates and thereby builds an abstract runtime model of the running multithreaded system. Analyzing this model on a level by level basis, the observer can infer effectively from *successful* execution of the observed system when basic safety properties can be violated by other executions. Attractive future work includes predictions of liveness violations and predictions of datarace and deadlock conditions.

7 Acknowledgments

The work is supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586, the DARPA IXO NEST Program, contract number F33615-01-C-1907), the ONR Grant N00014-02-1-0715, the Motorola Grant MOTOROLA RPS #23 ANT, and the joint NSF/NASA grant CCR-0234524.

References

1. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of Fifth International VMCAI con-*

ference (VMCAI'04) (To appear in LNCS), January 2004. Download: <http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp24.pdf>.

2. H. W. Cain and M. H. Lipasti. Verifying sequential consistency using vector clocks. In *Proceedings of the 14th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 153–154. ACM, 2002.
3. M. Dahm. Byte code engineering with the bcel api. Technical Report B-17-98, Freie Universitat Berlin, Institut fur Informatik, April 2001.
4. C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed debugging*, pages 183–194. ACM, 1988.
5. K. Havelund and G. Rou. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
6. K. Havelund and G. Rou. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
7. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
8. K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WADG'91)*, volume 579 of *Lecture Notes in Computer Science*, pages 254–272. Springer-Verlag, 1991.
9. F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1989.
10. A. Sen and V. K. .Garg. Partial order trace analyzer (pota) for distributed programs. In *Proceedings of the 3rd Workshop on Runtime Verification (RV'03)*, Electronic Notes in Theoretical Computer Science, 2003.
11. K. Sen and G. Rou. Generating optimal monitors for extended regular expressions. In *Proceedings of the 3rd Workshop on Runtime Verification (RV'03)*, volume 89 of *ENTCS*, pages 162–181. Elsevier Science, 2003.
12. K. Sen, G. Rou, and G. Agha. Java MultiPathExplorer (JMPaX 2.0). Download: <http://fsl.cs.uiuc.edu/jmpax>.
13. K. Sen, G. Rou, and G. Agha. Generating Optimal Linear Temporal Logic Monitors by Coinduction. In *Proceedings of 8th Asian Computing Science Conference (ASIAN'03)*, volume 2896 of *Lecture Notes in Computer Science*, pages 260–275. Springer-Verlag, December 2003.
14. K. Sen, G. Rou, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proceedings of 4th joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*. ACM, 2003.
15. K. Sen, G. Rou, and G. Agha. Runtime safety analysis of multithreaded programs. Technical Report UIUCDCS-R-2003-2334, University of Illinois at Urbana Champaign, April 2003.

A Flexible Coordination Framework for Application-Oriented Matchmaking and Brokering Services

Myeong-Wuk Jang, Amr M.E. Ahmed, Gul Agha
Open Systems Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign
{mjang, amrmomen, agha}@uiuc.edu

Abstract

An important problem for agents in open multiagent systems is how to find agents that match certain criteria. A number of middle agent services, such as matchmaking and brokering services, have been proposed to address this problem. However, the search capabilities of such services are relatively limited since the match criteria they use are relatively inflexible. We propose ATSpace, a middle agent to support application-oriented matchmaking and brokering services. Application agents in ATSpace deliver their own search algorithms to a public tuple space which holds agent property data; the tuple space executes the search algorithms on this data. We show how the ATSpace model increases the dynamicity and flexibility of middle agent services. Unfortunately, the model also introduces security threats: the data and access control restrictions in ATSpace may be compromised, and system availability may be affected. We describe some mechanisms to mitigate these security threats.

Keywords: Agent Coordination, Agent Interaction, Middle Agents, Brokering Services, Matchmaking Services.

1. Introduction

In multiagent systems, agents need to communicate with each other to accomplish their goals. An important problem in open multiagent systems is the *connection problem*: how to find agents that match given criteria [Dav83]. When agents are designed or owned by the same organization,

developers may be able to design agents which explicitly know the names of other agents that they need to communicate with. However in *open systems*, because different agents may dynamically enter or leave a system, it is generally not feasible to let agents know the names of all other agents that they need to communicate with at some point.

For solving the connection problem, Decker classifies middle agent services as either *matchmaking* (also called *Yellow Page*) services or *brokering* services [Dec96, Syc97]. Matchmaking services (e.g. Directory Facilitator in FIPA platforms [Fip02]) are passive services whose goal is to provide a client agent with a list of names of agents whose properties match its supplied criteria. The agent may later contact the matched agents to request services. On the other hand, brokering services (e.g. ActorSpace [Cal94]) are active services that directly deliver a message (or a request) to the relevant agents on their clients' behalf.

In both types of services, an agent advertises itself by sending a message which contains its name and a description of its characteristics to a *middle agent*. A middle agent may be implemented on top of a tuple space model such as Linda [Car89]; this involves imposing constraints on the format of the stored tuples and using Linda-supported primitives. Specifically, to implement matchmaking and brokering services on top of Linda, a tuple template may be used by the client agent to specify the matching criteria. However, the expressive power of a template is very limited; it consists of value constraints for its actual parameters and type constraints for its formal parameters. In order to overcome this limitation, Callsen's ActorSpace implementation used regular expressions in its search template [Agh93, Cal94]. Even though this implementation increased expressivity, its capability is still limited by the power of its regular expressions.

We propose *ATSpace*¹ (Active Tuple Spaces) to empower agents with the ability to provide arbitrary application-oriented search algorithms to a middle agent for execution on the tuple space. While ATSpace increases the dynamicity and flexibility of the tuple space model, it also introduces some security threats as codes developed by different groups with different interests are executed in the same space. We will discuss the implication of these threats and how they may be mitigated.

This paper is organized as follows. Section 2 explains the ATSpace architecture and introduces its primitives. Section 3 describes security threats occurred in ATSpace and addresses how to resolve them. Section 4 illustrates the power of the new primitives by describing experiments with using ATSpace on UAV (Unmanned Aerial Vehicle) simulations. Section 5 evaluates the

¹ We will use *ATSpace* to refer the model for a middle agent to support application-oriented service, while we use an *atSpace* to refer an instance of ATSpace.

performance of ATSpace and compares it with a general middle agent. Section 6 discusses related work, and finally, we conclude this paper with a summary of our research and future work.

2. ATSpace

2.1 A MOTIVATIVE EXAMPLE

We present a simple example to motivate the ATSpace model. In general, a tuple space user with a complex matching query is faced with the following two problems:

1. *Expressiveness problem* because a matching query cannot be expressed using the tuple space primitives.
2. *Incomplete information problem* because evaluating a matching query requires information which is not available for a tuple space manager.

For example, assume that a tuple space has information about seller agents and the prices of the products they sell; each tuple has the following attributes (`seller name`, `seller city`, `product name`, `product price`). Buyer agents can access the tuple space in order to find seller agents that sell, for instance, computers or printers. Also, a buyer agent wants to execute the following query:

*Q1: What are the **best two** (in terms of prices) sellers that offer computers and whose locations are roughly within 50 miles from me?*

A generic tuple space may not support the request of this buyer agent because, firstly, it may not support the “best two” primitive (problem 1), and secondly, it may not have information about the distance between cities (problem 2). Faced with these difficulties the buyer agent with the query Q1 has to transform it to a tuple template style query (Q2) to be accepted by the general tuple space. This query Q2 will retrieve a superset of the data that should have been retrieved by Q1.

Q2: Find all tuples about seller agents that sell computers.

The buyer agent then evaluates its own search algorithm on the returned data to find tuples that

satisfy Q1. In our example, the buyer agent would first filter out seller agents whose locations are less than 50 miles from the location of its user, and then choose the best two sellers from the remaining ones. To select seller agents located within 50 miles, the buyer agent has a way of estimating roughly distances between cities. Finally, it should send these seller agents a message to start the negotiation process.

An apparent disadvantage of the above approach is the movement of large amount of data from the tuple space to the buyer agent. When the tuple space includes large amount of tuples related to computer sellers, the size of the message to be delivered is also large. In order to reduce communication overhead, ATSpace allows a client agent to send an object containing its own search algorithm, instead of a tuple template. In our example, the buyer agent would send mobile code that inspects tuples in the tuple space and selects the best two sellers that satisfy the buyer criteria; the mobile code also carries information about distances to the near cities.

In Figure 1, the seller agents with AN2 and AN3 names are selected by the search algorithm, and the atSpace agent delivers `sendComputerBrand` message to them as a brokering service. Finally, the seller agents send information about brand names of their computers to the buyer agent.

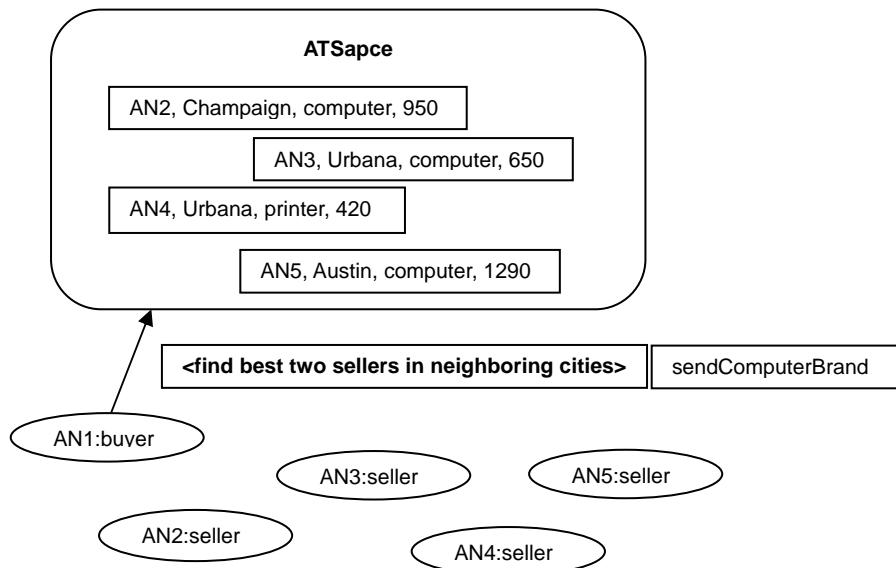


Figure 1: An Example of ATSpace

2.2 OVERALL ARCHITECTURE

ATSpace consists of three components: a tuple space, a message queue, and a tuple space manager (see Figure 2).

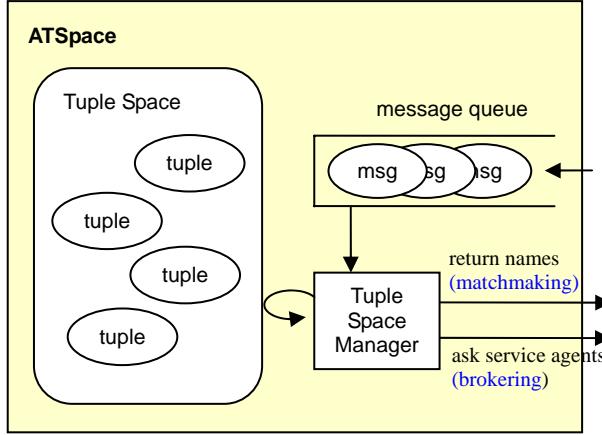


Figure 2: Basic Architecture of ATSpace

The tuple space is used as a shared pool for *agent tuples*, $\langle a, p_1, p_2, \dots, p_n \rangle$, which consists of a name field, a , and a property part, $P = p_1, p_2, \dots, p_n$ where $n \geq 1$; each tuple represents an agent whose name is given by the first field and whose characteristics are given by the subsequent fields. ATSpace enforces the rule that there cannot be more than one agent tuples whose agent names and property fields are identical. However, an agent may register itself with different properties (multiple tuples with the same name field), and different agents may register themselves with the same property fields (multiple tuples with the same property part).

$$\forall t_i, t_j : i \neq j \rightarrow [(t_i.a = t_j.a) \rightarrow (t_i.P \neq t_j.P) \& \& (t_i.P = t_j.P) \rightarrow (t_i.a \neq t_j.a)]$$

The message queue contains input messages that are received from other agents. Messages are classified into two types: *data input messages* and *service request messages*. A data input message includes a new agent tuple for insertion into the tuple space. A service request message includes either a tuple template or a mobile object. The template (or, alternately, the object) is used to search for agents with the appropriate agent tuples. A service request message may optionally contain another field, called the *service call message* field, to facilitate the brokering service. A *mobile*

object is an object that is provided by a service-requesting agent or client agent; such objects have pre-defined public methods, such as `find`. The `find` method is called by the tuple space manager with tuples in its `atSpace` as a parameter, and this method returns names of agents selected by the search algorithm specified in the mobile object.

The tuple space manager retrieves names of service agents whose properties match a tuple template or which are selected by a mobile object. In case of a matchmaking service, it returns the names to the client agent. In case of a brokering service, it forwards the service call message supplied by the client agent to the service agents.

2.3 OPERATION PRIMITIVES

General Tuple Space Primitives

The ATSpace model supports three basic primitives: `write`, `read`, and `take`. `write` is used to register an agent tuple into an `atSpace`, `read` is used to retrieve an agent tuple that matches a given criteria, and `take` is used to retrieve a matched agent tuple and remove it from the `atSpace`. When there are more than one agent tuples whose properties are matched with the given criteria, one of them is randomly selected by the agent tuple manager. When there is no a matched tuple, these primitives return immediately with an exception. In order to retrieve all agent tuples that match a given criteria, `readAll` or `takeAll` primitives should be used. The format² of these primitives is as follows:

```
void write(AgentName anATSpace, TupleData td);
AgentTuple read(AgentName anATSpace, TupleTemplate tt);
AgentTuple take(AgentName anATSpace, TupleTemplate tt);
AgentTuple[] readAll(AgentName anATSpace, TupleTemplate tt);
AgentTuple[] takeAll(AgentName anATSpace, TupleTemplate tt);
```

where `AgentName`, `TupleData`, `AgentTuple`, and `TupleTemplate` are data objects defined in ATSpace. A *data object* denotes an object that includes only methods to set and retrieve its member variables. When one of these primitives is called in an agent, the agent class handler creates a corresponding message and sends it to the `atSpace` specified as the first parameter, `anATSpace`. The `write` primitive causes a data input message while the others cause service request messages. Note that the `write` primitive does not include an agent tuple but a *tuple* that contains only the agent's

² The current ATSpace implementation is developed in the Java programming language, and hence, we use the Java syntax to express primitives.

properties. This is to avoid the case where an agent tries to register a property using another agent name to an atSpace. This tuple is then converted to an agent tuple with the name of the sender agent before the agent tuple is inserted to an atSpace.

In some applications, updating agent tuples happens very often. For such applications, *availability* and *integrity* are of great importance. Availability insures that at least one agent tuple exist at any time whereas integrity insures that old and new agent data do not exist simultaneously in an atSpace. Implementing the update request using two tuple space primitives, `take` and `write`, could result in one of these properties not being satisfied. If `update` is implemented using `take` followed by `write`, then availability is not met. On the other hand, if `update` is implemented using `write` followed by `take`, integrity is violated for a small amount of time. Therefore, ATSpace provides the `update` primitive to insure that `take` and `write` operations are performed as one atomic operation.

```
void update(AgentName anATSpace, TupleTemplate tt, TupleData td);
```

Matchmaking and Brokering Service Primitives

In addition, ATSpace also provides primitives for middle agent services: `searchOne` and `searchAll` for matchmaking services, and `deliverOne` and `deliverAll` for brokering services. Primitives for matchmaking are as follows:

```
AgentName searchOne(AgentName anATSpace, TupleTemplate tt);
AgentName searchOne(AgentName anATSpace, MobileObject ao);
AgentName[] searchAll(AgentName anATSpace, TupleTemplate tt);
AgentName[] searchAll(AgentName anATSpace, MobileObject ao);
```

The `searchOne` primitive is used to retrieve the name of a service agent that satisfies a given criteria, whereas the `searchAll` primitive is used to retrieve all names of service agents that match a given property.

Primitives for brokering service are as follows:

```
void deliverOne(AgentName anATSpace, TupleTemplate tt, Message msg);
void deliverOne(AgentName anATSpace, MobileObject ao, Message msg);
void deliverAll(AgentName anATSpace, TupleTemplate tt, Message msg);
void deliverAll(AgentName anATSpace, MobileObject ao, Message msg);
```

The `deliverOne` primitive is used to forward a specified service call message `msg` to the service agent that matches the given criteria, whereas the `deliverAll` primitive is used to send this message to all such service agents.

Note that our matchmaking and brokering service primitives allow agents to use *mobile objects* to support application-oriented search algorithm. We call matchmaking or brokering services used with mobile objects *active matchmaking* or *brokering services*. MobileObject is an abstract class that defines the interface methods between a mobile object and an atSpace. One of these methods is `find`, which may be used to provide the search algorithm to an atSpace. The format of the `find` method is defined as follows:

```
AgentName[] find(final AgentTuple[] ataTuples);
```

Service Specific Request Primitive

One drawback of the previous brokering primitives (`deliverOne` and `deliverAll`) is that they cannot support service-specific call messages. In some situations, a client agent cannot supply an atSpace with a service call message to be delivered to a service agent beforehand because it needs to examine the service agent properties first. Another drawback of the `deliverAll` primitive is that it stipulates that the same message should be sent to all service agents that match the supplied criteria. In some situations a client agent needs to send different messages to each service agent, depending on the service agent's properties. A client agent with any of the above requirements can use neither brokering services with tuple templates nor active brokering services with mobile objects. Therefore, the agent has to use the `readAll` primitive to retrieve relevant agent tuples and then create appropriate service call messages to send service agents selected. However, this approach suffers from the same problems as a general tuple space does.

To address the above shortcomings, we introduce the `exec` primitives. This primitive allows a client agent to supply a mobile object to an atSpace; the supplied mobile object has to implement the `doAction` method. When the method is called by an atSpace with agent tuples, it examines the properties of agents using the client agent application logic, creates different service call messages according to the agent properties, and then returns a list of *agent messages* to the atSpace to deliver the service call messages to the selected agents. Note that each agent message consists of the name of a service agent as well as a service call message to be delivered to the service agent. The formats of `exec` primitive and the `doAction` method are as follows.

```
void exec(AgentName anATSpace, MobileObject ao);
AgentMessage[] doAction(AgentTuple[] ataTuples);
```

3. SECURITY ISSUES

By allowing a mobile object to be supplied by an application agent, ATSpace supports application-oriented matchmaking and brokering services, which increases the flexibility and dynamicity of the tuple space model. However, it also introduces new security threats; we address some of these security threats and describe some ways to mitigate them. There are three important types of security issues for ATSpace:

- ◆ **Data Integrity:** A mobile object may not modify tuples owned by other agents.
- ◆ **Denial of Service:** A mobile object may not consume too much processing time or space of an atSpace, and a client agent may not repeatedly send mobile objects, thus overloading an atSpace.
- ◆ **Illegal Access:** A mobile object may not carry out unauthorized access or illegal operations.

We address the data integrity problem by blocking attempts to modify tuples. When a mobile object is executed by a tuple space manager, the manager makes a deep copy of tuples and then sends the copy to the `find` or `doAction` method of the mobile object. Therefore, even when a malicious agent changes some tuples, the original tuples are not affected by the modification. However, when the number of tuples in a tuple space is very large, this solution requires extra memory and computational resources. For better performance, the creator of an atSpace may select the option of delivering to mobile objects a shallow copy of the original tuples instead of a deep copy, although this will violate the integrity of tuples if an agent tries to delete or change tuples. We are currently investigating under what conditions a use of a shallow copy may be sufficient.

To address denial of service by consuming all processor cycles, we deploy user-level thread scheduling. Figure 3 depicts the extended architecture of ATSpace. When a mobile object arrives, the object is executed as a thread, and its priority is set to high. If the thread executes for a long time, its priority is continually downgraded. Moreover, if the running time of a mobile object exceeds a certain limit, it may be destroyed by the Tuple Space Manager; in this case, a message is sent to the sender agent of the mobile object to inform it about the destruction of the object. To incorporate these restrictions, we have extended the architecture of ATSpace by implementing job queues--thus making their semantics similar to that of actors. Other denial of service issues are still our on-going research.

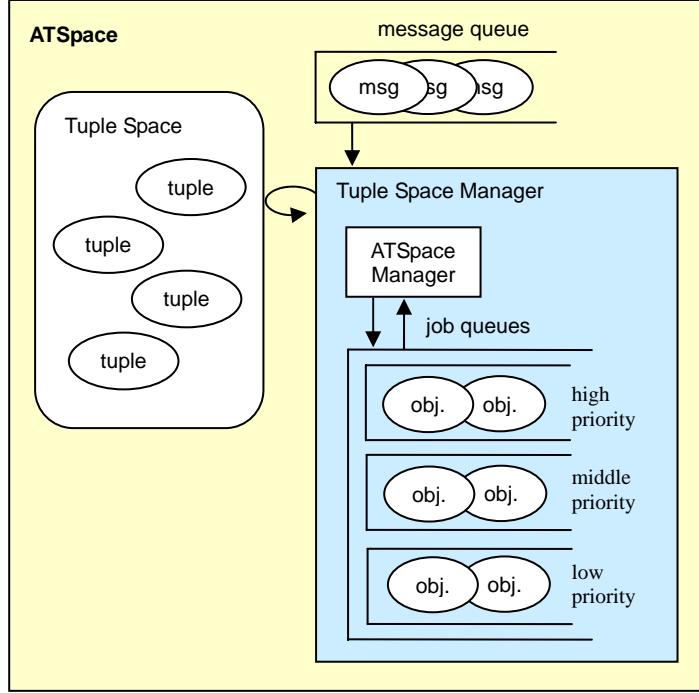


Figure 3: Extended Architecture of ATSpace

To prevent unauthorized access, an *atSpace* may be created with an *access key*; if an *atSpace* is created with an *access key*, then this key must accompany every message sent from service requester agents. Also, an *atSpace* may limit agents to modify only their own tuples.

4. Experiments

We have applied the ATSpace model in a UAV (Unmanned Aerial Vehicle) application which simulates the collaborative behavior of a set of UAVs in a surveillance mission [Jan03]. During the mission, a UAV needs to communicate with other neighboring UAVs within its local communication range (see Figure 4). We use the brokering primitives of ATSpace to accomplish this broadcasting behavior. Every UAV updates information about its location on an *atSpace* at every simulation step using the *update* primitive. When local broadcast communication is needed, the sender UAV (considered a client agent from the ATSpace perspective) uses the *deliverAll*

primitive and supplies as a parameter a mobile object³ that contains its location and communication range. When this mobile object is executed in the atSpace, the `find` method is called by the tuple space manager to find relevant receiver agents. The `find` method computes distances between the sender UAV and other UAVs to find neighboring ones within the given communication range. When the tuple space manager receives names of service agents, neighboring UAVs in this example, from the mobile object, it delivers the service call message given by the client agent--the sender UAV in this example--to them.

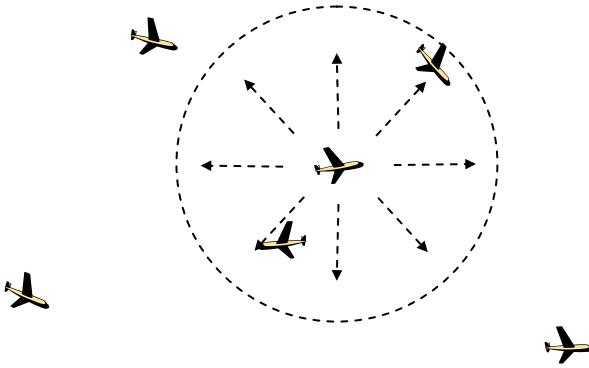


Figure 4: Simulation of Local Broadcast Communication

We also use ATSpace to simulate the behavior of UAV radar sensors. Each UAV should detect targets within its sensing radar range (see Figure 5). The *SensorSimulator*, which is the simulator component responsible for accomplishing this behavior, uses the `exec` primitive to implement this task. The mobile object⁴ supplied with the `exec` primitive computes distances between UAVs and targets, and decides neighboring targets for each UAV. It then creates messages each of which consists of the name of its receiver UAV and a service call message to be sent its receiver UAV agent. This service call message is simply the environment model around this UAV (neighboring targets in our domain). Finally, the mobile object returns these set of messages to the tuple space manager which in turn sends them to respective agents.

³ The code for this mobile object is in Appendix A.

⁴ The code for this mobile object is in Appendix B.

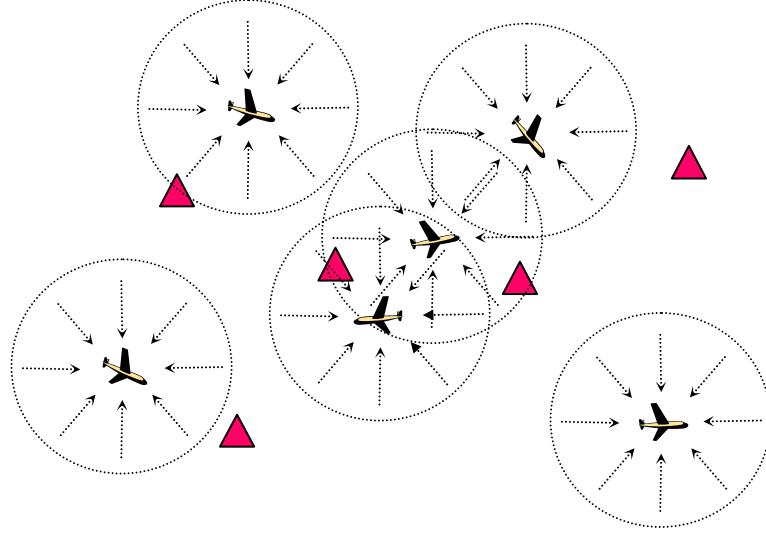


Figure 5: Simulation of Radar Sensor

5. Evaluation

The performance benefit of ATSpace can be measured by comparing its active brokering service with the data retrieval service of the template-based tuple space model along four different dimensions: the number of messages, the total size of messages, the total size of memory space on the clients' and middle agents' computers, and the time for the entire computation. To analytically evaluate ATSpace, we use the scenario described in section 2.1 where a service-requesting agent has a complex query that is not supported by the template-based model.

Let the number of service agents that satisfy this complex query be n . In the template-based tuple space model, the number of messages is $n + 2$. The details are as follows:

- ◆ $\text{Service_Request}_{\text{template}}$: a template-based service request message that includes Q2. A service-requesting agent sends this message to a tuple space to bring a superset of its final result.
- ◆ $\text{Service_Reply}_{\text{template}}$: a reply message that contains agent tuples satisfying Q2.
- ◆ $n \text{ Service_Call}$: n service call messages to be delivered by the service-requesting agent to the agents that match its original criteria Q1.

In ATSpace, the total number of messages is $n + 1$. This is because the service-requesting agent need not worry about the complexity of his query and only sends a service request message

(`Service_RequestATSpace`) to an atSpace. This message contains the code that represents its criteria along with a service call message which should be sent the agents that satisfy the criteria. The last n messages have the same explanation as in the template based model except that the sender is the atSpace instead of the service-requesting agent.

While the difference in the number of messages delivered in the two approaches is comparatively small, the difference in the total size of these messages may be huge. Specifically, the difference in bandwidth consumption (BD : Bandwidth Difference) between the template-based model and the ATSpace one is given by the following equation:

$$BD = [\text{size}(\text{Service_Request}_{\text{template}}) - \text{size}(\text{Service_Request}_{\text{ATSpace}})] + \text{size}(\text{Service_Reply}_{\text{template}})$$

In general the ATSpace service request message is larger, as it has the matching code, and thus the first component is negative. As such, ATSpace will only result in a bandwidth saving if the increase in the size of its service request message is smaller than the size of the service reply message in the template-based approach. This is likely to be true if the original query (Q1) is complex such that turning it into a simpler one (Q2) to retrieve a superset of the result would incur a great semantic loss and as such would retrieve a lot of the tuples from the template-based tuple space manager.

The amounts of the storage space used on the client agent's and middle agent's computers are similar in both cases. In the general tuple space, a copy of the tuples exists in the client agent, and an atSpace also requires a copy of the data for the mobile object to address the data integrity issue. However, if a creator of an atSpace opts to use a shallow copy of the data, the size of such a copy in the atSpace is much less than that of the copy in the client agent.

The difference in computation times of the entire operation in the two models depends on two factors: the time for sending messages and the time for evaluating queries on tuples. As we explained before, ATSpace will usually reduce the total size of messages so that the time for sending messages is in favor of ATSpace. Moreover, the tuples in the ATSpace are only inspected once by the mobile object sent by the service-requesting agent. However, in the template-based approach, some tuples are inspected twice: first, in order to evaluate Q2, the template-based tuple space manager needs to inspect all the tuples that it has, and second, the service-requesting agent inspects these tuples that satisfy Q2 to retain the tuples that also satisfy Q1. If Q1 is complex then Q2 may not filter tuples properly. Therefore, even though the time to evaluate Q2 against the entire tuples in the tuple space is smaller than the time needed to evaluate them by the mobile object, most

of the tuples on the tuple space manager may pass Q2 and be re-evaluated again by the service-requesting agent. This re-evaluation may have nearly the same complexity as running the mobile object code. Thus we can conclude that when the original query is complex and external communication cost is high, ATSpace will result in time saving.

Apart from the above analytical evaluation, we also evaluated the saving in computational time resulting from using the ATSpace in the UAV domain using the settings mentioned in section 4. Figure 6 shows the benefit of ATSpace compared to a general tuple space that provides the same semantic in the UAV simulation. In these experiments, UAVs use either active brokering service or data retrieval service to find their neighboring UAVs. In both cases, the middle agent includes information about locations of UAVs and targets. In case of the active brokering service, UAVs send mobile objects to the middle agent while UAVs using data retrieval service send tuple templates. The simulation time for each run is around 40 minutes, and the wall clock time depends on the number of agents. When the number of agents is small, the difference between the two approaches is not significant. However, as the number of agents is increased, the difference becomes large.

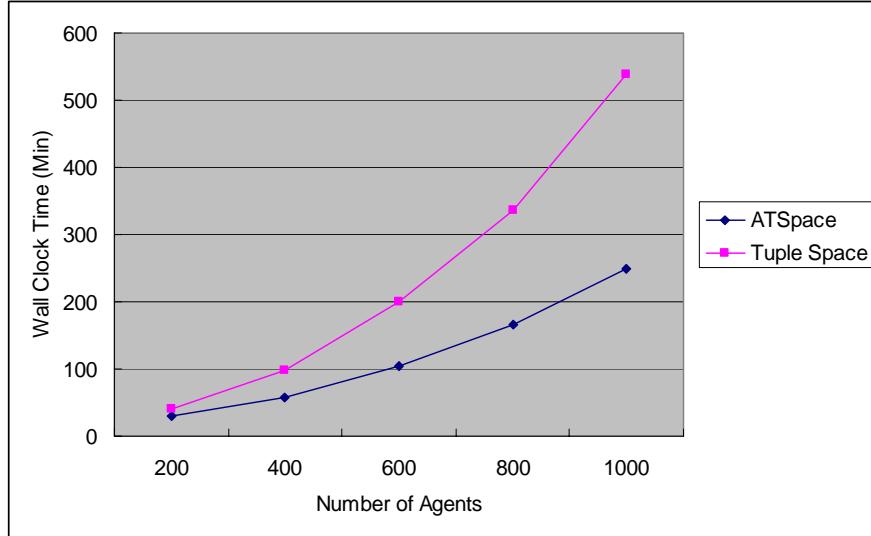


Figure 6: Wall Clock Time for ATSpace and Tuple Space

Figure 7 shows the number of messages, and Figure 8 shows the total size of messages in the two approaches, although the number of messages required is similar in both cases. However, a general tuple space requires more data movement than ATSpace, the shapes of these two lines in Figure 8 is similar to those in Figure 6. Therefore, we can hypothesize that there are strong

relationship between the total size of messages and the wall clock time of simulations.

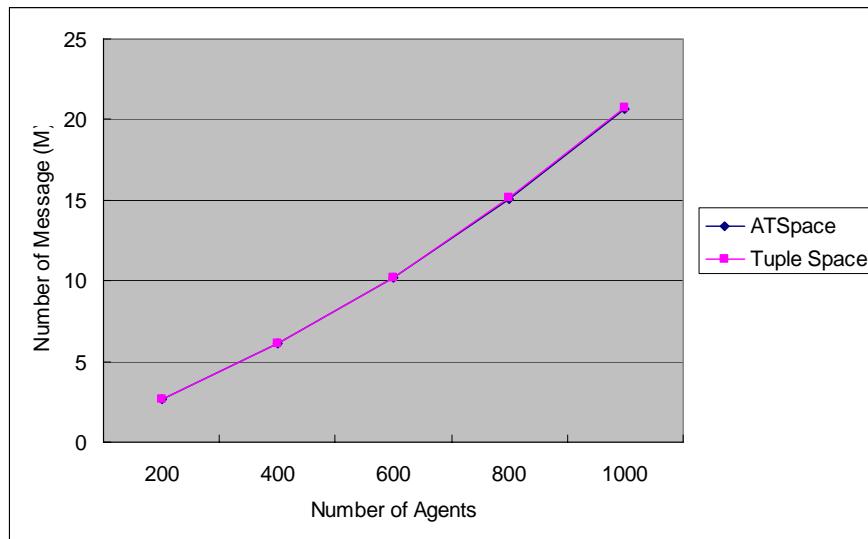


Figure 7: The Number of Messages for ATSpace and Tuple Space

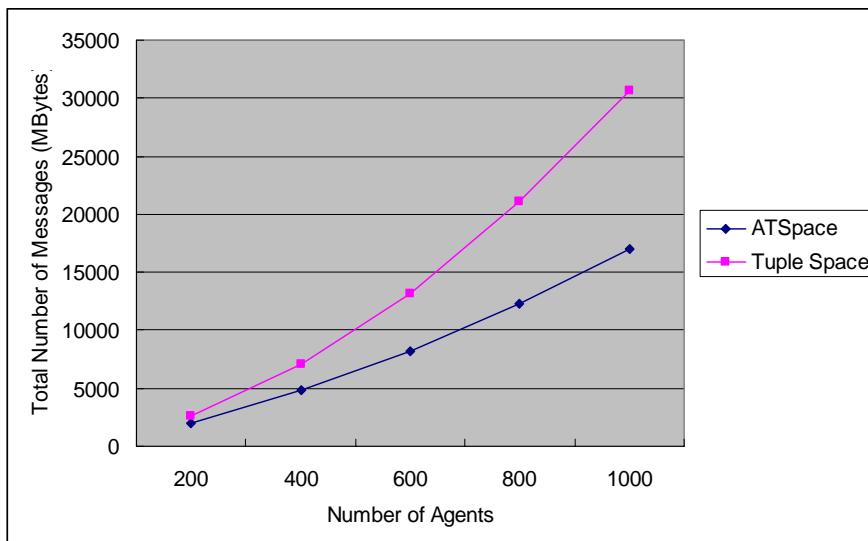


Figure 8: The Total Size of Messages for ATSpace and Tuple Space

6. Related Work

In this section we compare our ATSpace model with three related approaches: Other tuple space models, the Java Applet model, and finally mobile agents.

6.1 ATSpace Vs. Other Tuple Space Models

Our work is related to *Linda* [Car89, Gel85] and its variants, such as *JavaSpaces* and *TSpaces* [Leh99, Sun03]. In these models, processes communicate with other processes through a shared common space called a *blackboard* or a *tuple space* without considering references or names of other processes [Car89, Pfl98]. This approach was used in several agent frameworks, for example OAA and EMAF [Bae95, Mar97]. However, these models support only primitive features for anonymous communication among processes or agents.

From the middle agent perspective, *Directory Facilitator* in the *FIPA platform* and *Broker Agent* in *InfoSleuth* are related to our research [Fip02, Jac96]. However, these systems do not support customizable matching algorithm.

Some work has been done to extend the matching capability in the tuple space model. *Berlinda* allows a concrete entry class to extend the matching function [Tol97], and *TS* uses policy closures in a Scheme-like language to customize the behavior of tuple spaces [Jag91]. However, these approaches do not allow the matching function to be changed during execution. *OpenSpaces* provides a mechanism to change matching polices during execution [Duc00]. OpenSpaces groups entries in its space into classes and allows each class to have its individual matching algorithm. A manager for each class of entries can change the matching algorithm during execution. All agents that use entries under a given class are affected by any change to its matching algorithm. This is in contrast to ATSpace where each agent can supply its own matching algorithm without affecting other agents. Another difference between OpenSpaces and ATSpace is that the former requires a registration step before putting a new matching algorithm into action. *Object Space* allows distributed applications implemented in the C++ programming language to use a matching function in its template [Pol93]. This matching function is used to check whether an object tuple in the space is matched with the tuple template given in *rd* and *in* operators. However in ATSpace, the client agent supplied mobile objects can have a global overview of the tuples stored in the shared space, and hence, it can support global search behavior rather than one tuple based matching behavior supported in Object Space. For example, using ATSpace a client agent can find the best ten service agents according to its criteria whereas this behavior cannot be achieved in Object Space.

TuCSoN and *MARS* provide programmable coordination mechanisms for agents through Linda-

like tuple spaces to extend the expressive power of tuple spaces [Cab00, Omi98]. However, they differ in the way they approach the expressiveness problem; while TuCSoN and MARS use reactive tuples to extend the expressive power of tuple spaces, ATSpace uses mobile objects to support search algorithms defined by client agents. A reactive tuple handles a certain type of tuples and affects various clients, whereas a mobile object handles various types of tuples and affects only its creator agent. Also, these approaches do not provide an execution environment for client agents. Therefore, these may be considered as orthogonal approaches and can be combined with our approach.

6.2 The ATSpace Model vs. the Applet Model

ATSpace allows the movement of a mobile object to the ATSpace manager, and thus it can be confused with the Applet model. However, a mobile object in ATSpace quite differs from a Java applet: a mobile object moves from a client computer to a server computer while a Java applet moves from a server computer to a client computer. Also, the migration of a mobile object is initiated by its owner agent on the client computer, but that of a Java applet is initiated by the request of a client Web browser. Another difference is that a mobile object receives a method call from an atSpace agent after its migration, but a Java applet receives parameters and does not receive any method call from processes on the same computer.

6.3 Mobile Objects vs. Mobile Agents

A mobile object in ATSpace may be considered as a mobile agent because it moves from a client computer to a server computer. However, the behavior of a mobile object differs from that of a mobile agent. First of all, the behavior of objects in general can be compared with that of agents as follows:

- ◆ An object is *passive* while an agent is *active*, i.e., a mobile object does not initiate activity.
- ◆ An object does not have the *autonomy* that an agent has: a mobile object executes its method whenever it is called, but a mobile agent may ignore a request received from another agent.
- ◆ An object does not have a *universal name* to communicate with other remote objects; therefore, a mobile object cannot access a method on the remote object, but a mobile agent can communicate with agents on other computers. However, note that some object-based middleware may provide such functionality: e.g., objects in CORBA or DCOM [Vin97, Tha99]

may refer remote objects.

- ◆ The method interface of an object is precisely predefined, and this interface is directly used by a calling object.⁵ On the other hand, an agent may use a general communication channel to receive messages. Such messages require marshaling and unmarshaling, and have to be interpreted by receiver agents to activate the corresponding methods.
- ◆ While an object is executed as a part of a processor or a thread, an agent is executed as an independent entity; mobile objects may share references to data, but mobile agents do not.
- ◆ An object may use the reference passing in a method call, but an agent uses the value passing; when the size of parameters for a method call is large, passing the reference to local data is more efficient than passing a message, because the value passing requires a deep copy of data.

Besides the features of objects, we impose additional constraints on mobile objects in ATSpace:

- ◆ A mobile object can neither receive a message from an agent nor send a message to an agent.
- ◆ After a mobile object finishes its operation, the mobile object is destroyed by its current middle agent; a mobile object is used exactly once.
- ◆ A mobile object migrates only once; it is prevented from moving again.
- ◆ The identity of the creator of a mobile object is separated from the code of the mobile agent. Therefore, a middle agent cannot send a mobile object to another middle agent with the identity of the original creator of the object. Thus, even if the code of a mobile object is modified by a malicious server program, the object cannot adversely affect its creator. Moreover, since a mobile object cannot send a message to another agent, a mobile object is more secure than a mobile agent.⁶ However, a mobile object raises the same security issues for the server side.

In summary, a mobile object loses some of the flexibility of a mobile agent, but this loss is compensated by increased computational efficiency and security.

7. Conclusion and Future Work

In this technical report we presented ATSpace, Active Tuple Space, which works as a common shared space to exchange data among agents, a middle agent to support application-oriented

⁵ Methods of a Java object can be detected with the Java reflection mechanism. Therefore, the predefined interface is not necessary to activate a method of a Java object.

⁶ [Gre98] describes security issues of mobile agents in detail.

brokering and matchmaking services, and an execution environment for mobile objects utilizing data on its space. Our experiments with UAV surveillance simulations show that the model may be effective in reducing coordination costs. We have described some security threats that arise when using mobile objects for agent coordination, along with some mechanisms we use to mitigate them. We are currently incorporating memory use restrictions into the architecture and considering mechanisms to address denial of service attacks that may be caused by flooding the network [Shi02]. We also plan to extend ATSpace to support multiple tuple spaces distributed across the Internet (a feature that some Linda-like tuple spaces [Omi98, Sny02] already support).

Acknowledgements

This research is sponsored by the Defense Advanced Research Projects Agency under contract number F30602-00-2-0586.

References

- [Agh93] G. Agha and C.J. Callsen, “ActorSpaces: An Open Distributed Programming Paradigm,” *Proceedings of the 4th ACM Symposium on Principles & Practice of Parallel Programming*, pp. 23-32, May 1993.
- [Bae95] S. Baeg, S. Park, J. Choi, M. Jang, and Y. Lim, “Cooperation in Multiagent Systems,” *Intelligent Computer Communications (ICC ‘95)*, Cluj-Napoca, Romania, pp. 1-12, June 1995.
- [Cab00] G. Cabri, L. Leonardi, F. Zambonelli, “MARS: a Programmable Coordination Architecture for Mobile Agents,” *IEEE Computing*, Vol. 4, No. 4, pp. 26-35, 2000.
- [Cal94] C. Callsen and G. Agha, “Open Heterogeneous Computing in ActorSpace,” *Journal of Parallel and Distributed Computing*, Vol. 21, No. 3, pp. 289-300, 1994.
- [Car89] N. Carreiro, and D. Gelernter, “Linda in context,” *Communications of the ACM*, Vol. 32, No. 4, pp. 444-458, 1989.
- [Dav83] R. Davis and R.G. Smith, “Negotiation as a Metaphor for Distributed Problem Solving,” *Artificial Intelligence*, Vol. 20, No. 1, pp. 63-109, January 1983.
- [Dec96] K. Decker, M. Williamson, and K. Sycara, “Matchmaking and Brokering,” *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS-96)*, Kyoto, Japan, December, 1996.

[Duc00] S. Ducasse, T. Hofmann, and O. Nierstrasz, “OpenSpaces: An Object-Oriented Framework for Reconfigurable Coordination Spaces,” In A. Porto and G.C. Roman, (Eds.), *Coordination Language and Models, LNCS 1906*, Limassol, Cyprus, pp. 1-19, September 2000.

[Fip02] Foundation for Intelligent Physical Agents, *SC00023J: FIPA Agent Management Specification*, December 2002. <http://www.fipa.org/specs/fipa00023/>

[Gel85] D. Gelernter, “Generative Communication in Linda,” *ACM Transactions on Programming Language and Systems*, Vol. 7, No. 1, pp. 80-112, January 1985.

[Gre98] M.S. Greenberg, J.C. Byington, and D.G. Harper, “Mobile Agents and Security,” *IEEE Communications Magazine*, Vol. 36, No. 7, pp. 76-85, July 1998.

[Jac96] N. Jacobs and R. Shea, “The Role of Java in InfoSleuth: Agent-based Exploitation of Heterogeneous Information Resources,” *Proceedings of Intranet-96 Java Developers Conference*, April 1996.

[Jag91] S. Jagannathan, “Customization of First-Class Tuple-Spaces in a Higher-Order Language,” *Proceedings of the Conference on Parallel Architectures and Languages - Vol. 2, LNCS 506*, Springer-Verlag, pp. 254-276, 1991.

[Jan03] M. Jang, S. Reddy, P. Tasic, L. Chen, and G. Agha. “An Actor-based Simulation for Studying UAV Coordination,” *Proceedings of the 15th European Simulation Symposium (ESS 2003)*, pp. 593-601, October 2003

[Leh99] T.J. Lehman, S.W. McLaughry, and P. Wyckoff, “TSpaces: The Next Wave,” *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS-32)*, January 1999.

[Mar97] D.L. Martin, H. Oohama, D. Moran, and A. Cheyer, “Information Brokering in an Agent Architecture,” *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, London, pp. 467-489, April 1997.

[Omi98] A. Omicini and F. Zambonelli, “TuCSoN: a Coordination Model for Mobile Information Agents,” *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, Pisa, Italy, June 1998.

[Pfl98] K. Pfleger and B. Hayes-Roth, *An Introduction to Blackboard-Style Systems Organization*, KSL Technical Report KSL-98-03, Stanford Knowledge Systems Laboratory, January 1998.

[Pol93] A. Polze, “Using the Object Space: a Distributed Parallel make,” *Proceedings of the 4th*

IEEE Workshop on Future Trends of Distributed Computing Systems, Lisbon, pp. 234-239, September 1993.

[Shi02] C. Shields, “What do we mean by Network Denial of Service?,” *Proceedings of the 2002 IEEE Workshop on Information Assurance and Security*, United States Military Academy, West Point, NY, pp. 17-19, June 2002.

[Sny02] J. Snyder and R. Menezes, “Using Logical Operators as an Extended Coordination Mechanism in Linda,” In F. Arbab and C. Talcott (Eds.), *Coordination 2002, LNCS 2315*, Springer-Verlag, Berlin Heidelberg, pp. 317-331, 2002.

[Sun03] Sun Microsystems, *JavaSpacesTM Service Specification*, Ver. 2.0, June 2003.
<http://java.sun.com/products/jini/specs>

[Syc97] K. Sycara, K. Decker, and M. Williamson, “Middle-Agents for the Internet,” *Proceedings of the 15th Joint Conference on Artificial Intelligences (IJCAI-97)*, pp. 578-583, 1997.

[Tha99] T.L. Thai, Learning DCOM, O'Reilly & Associates, 1999.

[Tol97] R. Tolksdorf, “Berlinda: An Object-oriented Platform for Implementing Coordination Language in Java,” *Proceedings of COORDINATION '97 (Coordination Languages and Models, LNCS 1282*, Pringer-Verlag, pp. 430-433, 1997.

[Vin97] S. Vinoski, “CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments,” *IEEE Communications*, Vol. 14, No. 2, pp. 46-55, Feb 1997.

Appendix A: Mobile Object for Local Broadcast Communication

```
public class CommunicationMobileObject implements MobileObject
{
    protected final static double BROADCAST_RANGE = 50000.0;
        // range for broadcast communication

    private Point m_poPosition;
        // location of the current location of a UAV

    /**
     * Creates a mobile object with the location of the caller UAV agent.
     */
    public CommunicationMobileObject(Point p_point)
    {
        m_poPosition = p_point;
    }

    /**
     * Defines the 'find' method.
     */
    public AgentName[] find(AgentTuple[] p_ataTuples)
    {
        double dEWDistance = m_poPosition.getX();
        double dNSDistance = m_poPosition.getY();

        LinkedList llReceivers = new LinkedList();

        for (int i=0; i<p_ataTuples.length; i++) {
            if (p_ataTuples[i].sizeOfElements() == 1) {
                Object objItem = p_ataTuples[i].getElement(0);

                try {
                    //
                    // check the type of a field of a tuple.
                    //
                    if ( (Class.forName("app.task.uav.Point")).isInstance(objItem)) ) {
                        Point poObject = (Point) objItem;
                        double dDistance =
                            Math.sqrt( Math.pow(poObject.getX() - dEWDistance, 2.0) +
                                       Math.pow(poObject.getY() - dNSDistance, 2.0) );

                        //
                        // compute the distance between the caller UAV and another.
                        //
                        if ( dDistance <= BROADCAST_RANGE) {
                            llReceivers.add(p_ataTuples[i].getAgentName());
                        }
                    }
                } catch (ClassNotFoundException e) {
                    System.err.println(">> Investigator.search: " + e);
                }
            }
        }

        //
        // return the names of neighboring UAV agents.
        //
        AgentName anaReceivers[] = new AgentName[llReceivers.size()];
        llReceivers.toArray(anaReceivers);
        return anaReceivers;
    }
}
```

Appendix B: Mobile Object for Sensors of UAVs.

```

public class SensorMobileObject implements MobileObject
{
    public final static double RADAR_SENSOR_RANGE = 25000.0;
        // range for radar sensing

    private final static double RADAR_SENSOR_ALTITUDE = 2000.0;
        // minimum altitude of a UAV to detect an object by a radar

    private ObjectInfo[] m_oiaNeighboringObject;

    /**
     * Defines the 'doAction' method.
     */
    public AgentMessage[] doAction(final AgentTuple[] p_ataAgentTuples)
    {
        LinkedList llMsgs = new LinkedList();

        //
        // classify the tuples into UAVs or Targets.
        //
        LinkedList llUAVs = new LinkedList();
        LinkedList llTargets = new LinkedList();

        for (int i=0; i<p_ataAgentTuples.length; i++) {
            if (p_ataAgentTuples[i].sizeOfElements() == 1) {
                try {
                    Object objItem = p_ataAgentTuples[i].getElement(0);

                    //
                    // check the type of a field of a tuple.
                    //
                    if ( (Class.forName("app.task.uav.Point")).isInstance(objItem)) {
                        //
                        // if a UAV is lower than the predefined minimum altitude,
                        // then ignore the UAV.
                        //
                        Point poUAV = (Point) objItem;
                        if (poUAV.getZ() >= RADAR_SENSOR_ALTITUDE) {
                            llUAVs.add(p_ataAgentTuples[i]);
                        }
                    } else if (Class.forName("app.task.uav.ObjectInfo").isInstance(objItem)) {
                        ObjectInfo oiTarget = (ObjectInfo) objItem;
                        llTargets.add(oiTarget);
                    }
                } catch (ClassNotFoundException e) {
                    System.err.println(">>SensorMobileObject.doAction: " + e);
                }
            }
        }

        //
        // change LinkedList-type data to Array-type data.
        //
        AgentTuple[] ataUAVs = new AgentTuple[llUAVs.size()];
        llUAVs.toArray(ataUAVs);

        ObjectInfo[] oiaTargets = new ObjectInfo[llTargets.size()];
        llTargets.toArray(oiaTargets);

        //
        // compute horizontal distance and vertical distance among UAVs.
        //
        m_oiaNeighboringObject = new ObjectInfo[oiaTargets.length];
    }
}

```

```

for (int i=0; i<ataUAVs.length; i++) {
    //
    // collect neighboring objects, such as targets.
    //
    int iNumNeighboringObjects = 0;

    Point pointUAV = (Point) ataUAVs[i].getElement(0);
    double dX = pointUAV.getX();
    double dY = pointUAV.getY();

    for (int j=0; j<oiaTargets.length; j++) {
        double dDistance =
            java.lang.Math.sqrt(
                Math.pow(dX - oiaTargets[j].getEWDistance(), 2.0) +
                Math.pow(dY - oiaTargets[j].getNSDistance(), 2.0) );

        if ( (i != j) &&
            (dDistance < RADAR_SENSOR_RANGE) ) {
            oiaTargets[j].setHDistance(dDistance);
            oiaTargets[j].setVDistance(0);

            m_oiaNeighboringObject[iNumNeighboringObjects++] = oiaTargets[j];
        }
    }

    //
    // if there are more than one neighboring objects,
    // then create a message to send information about them to the UAV.
    //
    if (iNumNeighboringObjects > 0) {
        ObjectInfo[] oiaObjectDetected = new ObjectInfo[iNumNeighboringObjects];

        System.arraycopy(m_oiaNeighboringObject, 0,
                        oiaObjectDetected, 0,
                        iNumNeighboringObjects);

        Object[] objArgs = { oiaObjectDetected };

        llMsgs.add(createAgentMessage(ataUAVs[i].getAgentName(), "alarm", objArgs));
    }
}

//
// return agent messages to ATSpace.
//
AgentMessage anaMsgs[] = new AgentName[llMsgs.size()];
llReceivers.toArray(anaMsgs);
return anaMsgs;
}
}

```

A Perspective on the Future of Massively Parallel Computing: Fine-Grain vs. Coarse-Grain Parallel Models

Comparison & Contrast

Predrag T. Tosic

Open Systems Laboratory, Department of Computer Science,
University of Illinois at Urbana Champaign (UIUC)
201 N. Goodwin, Urbana, IL 61801 USA
Email: p-tosic@cs.uiuc.edu Fax: 217 - 333 - 9386

ABSTRACT

Models, architectures and languages for *parallel computation* have been of utmost research interest in computer science and engineering for several decades. A great variety of parallel computation models has been proposed and studied, and different parallel and distributed architectures designed as some possible ways of harnessing parallelism and improving performance of the general purpose computers.

Massively parallel abstract connectionist models such as *artificial neural networks (ANNs)* and *cellular automata (CA)* have been primarily studied in domain-specific contexts, namely, *learning* and *complex dynamics*, respectively. However, they can also be viewed as generic abstract models of massively parallel computers that are in many respects fundamentally different from the “main stream” parallel and distributed computation models.

We compare and contrast herewith the parallel computers as they have been built by the engineers with those built by Nature. We subsequently venture onto a high-level discussion of the properties and potential advantages of the proposed massively parallel computers of the future that would be based on the fine-grained connectionist parallel models, rather than on either various multiprocessor architectures, or networked distributed systems, which are the two main architecture paradigms in building parallel computers of the late 20th and early 21st centuries. The comparisons and contrasts herein are focusing on the fundamental conceptual characteristics of various models rather than any particular engineering idiosyncrasies, and are carried out at both *structural* and *functional* levels. The fundamental distinctions between the fine-grain connectionist parallel models and their “classical” coarse-grain counterparts are discussed, and some important expected advantages of the hypothetical massively parallel computers based on the connectionist paradigms conjectured.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CF'04, April 14–16, 2004, Ischia, Italy.

Copyright 2004 ACM 1-58113-741-9/04/0004 ...\$5.00.

We conclude with some brief remarks on the role that the paradigms, concepts, and design ideas originating from the connectionist models have already had in the existing parallel design, and what further role the connectionist models may have in the foreseeable future of parallel and distributed computing.

Categories and Subject Descriptors

A.1 [General]: Introductory and Survey; B.6.1 [Logic Design]: Cellular Arrays and Automata; C.1.3; C.1.4; C.2.4 [Processor Architectures]: Cellular Architectures, Neural Nets; Parallel Architectures; Distributed Systems; F.1.4 [Computation by Abstract Devices]: Models of Computation

General Terms

Design, Theory

Keywords

parallel computation models, massively parallel computing, multiprocessor computers, distributed systems, neural networks, cellular automata

1. INTRODUCTION AND BACKGROUND ON PARALLEL MODELS

Finding, utilizing and creating *parallelism* in computing have been major subjects of research, as well as among the grandest challenges for computer software and hardware designers alike since at least the early 1980s. The primary driving force behind this quest for (preferably, massive) parallelism in both software and hardware has been, and still is, the never-ending need for yet higher performance and, in particular, primarily for higher *computational speed*.

Namely, there are certain fundamental limitations, imposed by the laws of physics, to *how fast* one can compute with a strictly sequential computation. These limitations, by their very nature, are technology - independent. To appreciate the urge for genuinely new computation models, however, it is worth while to consider the existing technologies, and why the mere further *evolution* along the well-established paths in computer hardware and architecture

designs *alone* are not going to be able to keep sustaining the exponential rate computer performance improvements for longer than another decade or two (at the most). To motivate what is to follow, let us, therefore, briefly consider some existing technologies.

It has been argued that the two main technological driving forces at the hardware level behind sustaining the exponential growth in digital computer speeds over the span of several decades have been (i) the continued power-law improvements in increasing the densities and decreasing the sizes of the basic micro-components such as switching circuits, and (ii) similar continued improvements in the clock cycle speeds [AKFG]. While the ultimate limits have not been reached (yet), the fundamental laws of physics imply that these limits are finite and, therefore, will eventually be reached. More specifically, in case of the ever-decreasing sizes of the basic electronic components, it is the laws of micro-universe based on principles of quantum mechanics that pose the fundamental limits to just how small (and how dense on a die or a chip) can these basic electronic components be. For instance, with component sizes approaching or reaching the nano-scale and beyond, the quantum mechanical effects and fundamental principles such as, e.g., Heisenberg's uncertainty principle, become of utmost importance, as even the theoretical possibility of full physical controllability of these nano-devices and their behavior becomes a major issue. In case of clock cycles, on the other hand, it is another fundamental law of physics - the principle from special relativity that no signal can propagate faster than the speed of light - that implies that the power-law improvements in processor speeds simply cannot continue *ad infinitum*.

It is important to emphasize that these ultimate physical limits are *fundamental*, and therefore independent of any particular (current or future) computer hardware technology or implementation. It is immediate that one can keep decreasing the size of semiconductor transistors, or making the CPUs as well as memories faster, but only to within the limitations imposed by the laws of physics. However, it is important to understand and appreciate that the implications of, e.g., finite speeds of *any kind of signal* propagation in *any kind of a physical system or medium* (computational or otherwise), will still apply to any conceivable future computer designs, irrespective of how profoundly different these information processing machines or media of the future may be from our past or present designs. In particular, these fundamental physical limits will unavoidably have some profound implications on the computer technologies of the future, even if, for example, the new technologies do not use semiconductor transistors (or other types of transistors) at all, or even if those computers do not have CPUs and memories as we have known them since the very advent of electronic digital computers.

Of all the physical limits imposed by the laws of Nature, we find the finite speed of information propagation to have the most profound and far-reaching implications. This alone should suffice to start seriously considering computing models of *as massive parallelism (at all levels, and, in particular, in the hardware) as possible*, that is, as massive as the *Nature* permits. In this quest for such massively parallel models, however, one should not limit his explorations only to those parallel computing models, designs and/or implementations that share a good deal of similarity with the known, existing technologies, and could therefore be viewed as (possible) further *evolutionary* advancements along the

well-known general principles. In particular, the current (or anticipated near future) state of computer engineering and technologies should not impose a limit on our vision of the conceivable massively parallel information processing devices and media of the future. This paper is a modest attempt to indicate some possible early steps in such a pursuit of the truly *revolutionary* models of parallel computing, and of the new parallel computing frontiers.

1.1 Limits of the “von-Neumannian” sequential models

The classical computer design, also known as “*von Neumann computer architecture*”, is based on the following two premises: (i) there is a clear physical as well as logical separation from where the data and programs are stored (memory), and where the computation is executed (processor(s)); and (ii) a processor executes basic instructions (operations) one at a time, i.e., sequentially. Many partial departures have been made from this basic sequential computer model. For example, pipelining is one of the ways of exploiting *instruction-level parallelism (ILP)* even in a single-processor machine. Yet, the fundamental distinction between processing and storage, for the most part¹, has remained. In particular, (i) implies that data has to travel from where it is stored to where it is processed (and back), and it can only do so at a speed no greater than the speed of light, whereas (ii) implies that the basic instructions, including fetching the data from or returning the data to the storage, are, beyond some benefits due to internal structure and modularity of processors, and the possibility of exploiting the aforementioned instruction-level parallelism², essentially still executed one at a time - and each such instruction has to take some finite, however minuscule, amount of time. We motivate the introduction of various (classical) parallel models as a way of overcoming the limitations of “one instruction at a time” later in this section, and study the alternatives to the existing processor vs. memory duality in some detail in §2 and §3.

In particular, we emphasize again that *any* sequential computer, von Neumann model based or otherwise, that consists of a single processing element capable of executing only one instruction at a time at some finite speed, however fast, is eventually doomed to lose the battle with ever-more demanding and more complex applications and their performance requirements³. Hence, building computers that have multiple processing elements was realized to be a matter of necessity many years ago.

Some of the fundamental questions that naturally arise in this context are the following:

- On what basic premises should massively parallel systems be built?

¹However, see §§3.1 and §4.

²Further discussion on (ILP) and similar techniques are beyond our scope herein. It suffices to say, that ILP may considerably improve the sequential computer's performance, but it cannot fundamentally overcome the bottlenecks stemming from these two fundamental assumptions.

³In addition, there are so many important computational problems that are, besides the issue of efficiency, much more elegantly solved in parallel, and, in particular, can be naturally mapped onto an architecture with multiple processors. However, as already pointed out, it is chiefly the performance, and among various performance metrics, primarily the speed of execution, that have been the dominant driving forces behind the quest for massive parallelism.

- How are the different processing units to coordinate their actions and split up among themselves the computation and the needed resources?

- What are the ultimate limits of the computational power of these parallel models, and what are the limits on how much can be gained with these models in terms of speed and other performance metrics, in comparison to the limitations and resource requirements of comparable sequential models (see, e.g., [GISP], [PAPA])?

This work focuses on comparison and contrast of two profoundly different classes of parallel models. These two classes of models are based on fundamentally different basic premises, and their architectures are, consequently, entirely different. The first class are the “classical” parallel computers, such as multiprocessor supercomputers and networked distributed systems⁴ as we know them. This class and their main conceptual characteristics are the subject of the rest of §1. The second class are those *fine-grained parallel models* that are based on the *connectionist* paradigm [GARZ]. The two perhaps best known examples of this, connectionist kind of fine-grained parallelism, viz., cellular automata and neural networks, will be introduced in section §2.

We make an important clarification regarding our use of the terms *fine-grain(ed)* and *coarse-grain(ed)* parallelism. First, traditional computer science distinguishes fine vs. coarse granularity of parallel computing at both software and hardware levels. For example, at the software level, one can discuss fine-grain vs coarse-grain code decomposition (into, e.g., objects). There are also appropriate notions of granularity, and therefore fine- and coarse-grained parallelism, when it comes to parallel programming and optimizing compiler techniques; an interested reader is referred to, e.g., [ALKE]. We do not concern ourselves herein with any of those notions of granularity. Instead, we focus on the notion of granularity, and compare fine vs. coarse parallelism, entirely in the context of (abstract) models of parallel computer architectures. Moreover, our notion of fine-grained parallelism (in the hardware) is based on the connectionist paradigm [GARZ], whereas by coarse-grained parallelism we mean all classical multiprocessor, multicomputer and distributed systems - including those that much of the literature would label “fine-grain”, such as, e.g., distributed memory multiprocessor supercomputers that may have thousands or even tens of thousands of processors. In particular, by “*fine-grain(ed) parallel models*” we shall heretofore mean, *connectionist fine-grain(ed) massively parallel models*.

Regarding the issue of the choice of an architecture of massively parallel machines, which spans the first two questions in the list above, and has important implications for the third, some crucial design challenges include:

- How should the multiple processors be organized and interconnected together?
- Are these processors to share storage, or is each of them

⁴We primarily focus on these two existing architecture models for parallel computation, as they are, in a sense, the two “extremes” and also the two “purest” parallel computer paradigms. There are many other, “intermediate” models, such as, e.g., distributed memory multi-processor models, that fall “somewhere in between” the shared-memory multiprocessors and the (physically as well as logically) distributed collections of computers or computing systems, that are then all collected into a single distributed system by an appropriate *communication network*, and that we shall simply refer to as *distributed systems*.

- to have its own “local” memory?

- How tight should be the coupling among different processors (and possibly other components of such a massively parallel computing system)?

- How are different processors to communicate and coordinate with one another?

Each of the main design choices for these dilemmas yields a distinct class of parallel models with its specific paradigms and features. For example, the design choice of having a shared storage leads to various shared memory models, where the processors tend to be tightly coupled together, and where the main (possibly only) way different processors communicate with one another is via reading from and writing to the shared memory. Similarly, the choice of the coordination model leads to models where communication is (justifiably assumed) synchronous on one, and those where communication has to be treated as asynchronous, on the other hand. These two categories of models often require considerably different techniques to design and analyze, different algorithmic paradigms, different models to formally reason about, etc.

In the rest of this section, we outline two main classes of the parallel computer models in computer science and computer engineering of yesterday and today. These two classes are:

- (i) tightly coupled multiprocessor systems (that typically have shared memory or, at least, shared address space); and

- (ii) loosely coupled, message passing based distributed systems (that share neither memory nor the address space).

While fundamentally different in many respects, these two classes of parallel models also share some important features. One of them is that in both cases each individual processor is computationally powerful. An altogether different approach to building massively parallel systems could be based, instead, on very-large-size networks or grids of interconnected simple processing units. The power of parallel systems that would be built on this basic premise would primarily stem from the complex interconnection and interaction patterns among entities that are by themselves of a very limited computational power. The “granularity” of this alternative kind of parallel models would therefore be considerably finer than that of the standard multiprocessor or networked distributed system (or other “classical” parallel) architectures. For this reason, the two particular examples of classes of models studied in this section, viz., the shared-memory multiprocessors and the message-passing distributed systems, we also call *models of coarse-grained parallelism*, in contrast to their alternative, that is, *the connectionist models of fine-grained parallelism*, that will be studied in §2.

1.2 Tightly coupled multiprocessors

We begin the discussion of parallel computing models by first considering the classical *von Neumann architecture* with a single processor and a memory, and some link (i.e., data path) between the two. How is this architecture to be extended if we are to add more processors?

There are at least two obvious choices. One is, to have several processors, each with its own link to the one, *shared (or global) memory*. Architectures based on this generic model are called *shared memory machines*. In particular, processors communicate with one another via reading from and writing to the shared memory.

The second obvious choice is that each processor has its own individual memory. Since, unless connected, these dif-

ferent processor + memory pairs would really be distinct, independent computers rather than a single computing system, there has to be a common link, usually called *the bus*, that connects all processors together. Hence, each processor does its own computation and can directly access only its own, local memory, but processors can also communicate with one another via the shared bus. Parallel architectures based on this general idea are called *distributed memory machines*.

The two models outlined above can be viewed as “pure” shared and distributed memory models, at least insofar as the multi-processor parallel architecture designs are concerned. Many hybrid variants readily come to one’s mind. In particular, in a typical shared memory multiprocessor one finds that each processor, in addition to the global shared memory, also has its own, local memory. This local memory of each processor can be just a cache of relatively small size, or a sizable “main” memory, possibly with its own partial or full-fledged memory hierarchy.

Both the shared memory architectures and the distributed memory architectures based on the general premises outlined above are examples of *multiprocessor* parallel computers. In multiprocessor machines, regardless of whether the memory is shared or distributed (or “a little bit of both”), most or all of other devices and resources comprising a computer system are shared. Moreover, not only are, e.g., input-output (I/O) devices shared by all the processors, but also the operating system (OS), as an interface between the hardware and the (software) applications run on that hardware, is one and the same for all processors.

While the memory organization is one of the central design choices that various classifications and “taxonomies” of the standard parallel models are usually based upon (e.g., [XAIY]), there are many other design parameters where different design choices lead to significantly different parallel architectures, that, in their turn, have considerable implications for the *functionality*, i.e., the ways parallel processing is carried out, in these different models. Some of these critical design parameters include:

- *communication “medium”*: shared memory vs. distributed memory, message passing systems;
- *the nature of memory access*: uniform vs. nonuniform;
- *granularity*: a very large number of simple processing units vs. relatively small number of powerful processors;
- *instruction streams*: single vs. many;
- *data streams*: single vs. many; etc.

For example, if there is a single instruction stream, so that each processor in a multi-processor machine executes the same instructions as all other processors, yet different processors execute these instructions on different data, then we get a *Single Instruction, Multiple Data (SIMD)* model. *MIMD* and *SISD* models are defined similarly.

There are many fundamental challenges (to be distinguished from mere engineering limitations or deficiencies of the past or present technologies) that designers of a tightly coupled multiprocessor have to overcome in order to achieve *effective massive parallelism* and, consequently, the desired system performance, resource sharing, etc. In case of the shared memory machines, the most important problem is, how to achieve *scalability* as the number of processors and the size of the shared memory grow. For more on shared memory models, their strengths and deficiencies, the reader is referred to the literature about *Parallel Random Access Machines* and related shared memory abstract models; two

such references are [KARA] and [GISP]. The trends in computer technology have already validated the distributed memory parallel models to be more scalable than their shared memory counterparts. In case of distributed memory multiprocessors, it is the coordination and communication via *message passing* that pose some of the major challenges - together with the fact of life that such systems are more difficult to both program and formally reason about.

1.3 Loosely coupled distributed systems

The multiprocessor approaches outlined in the previous subsection are not the only way of achieving parallelism at the hardware level. Another possibility is to have a collection of different computers, with possibly quite heterogeneous architectures, operating systems etc., and then to interconnect all these distinct machines with an interconnection network, so that they can communicate and cooperate with each other. Clearly there would be no shared memory in such models - moreover, there is no “shared bus”, either, and each individual machine (also called a *node*) in this *multicomputer* model has its own peripheral devices, its OS may differ from those of other nodes, etc. In order for such a heterogeneous collection of computers to be able to function in unison, a common interface, called *middleware*, needs to be provided. Once there is a network connecting all these different computers, and a common middleware interface so that they can “understand each other”, different machines in such a multicomputer system interact with one another via *sending and receiving messages*. This exchange of messages may be taking place across the considerable physical distances - this is the case in systems where different computers are connected by the *wide-area networks* (WANs). Multicomputer systems of various sorts that are interconnected by a communication network and whose design is along the general lines that we have just outlined are also known as *distributed systems*⁵.

Distributed systems have several advantages over the multiprocessor machines. The chief advantage is that, overall, they scale better. They are also more flexible, more readily modifiable, reconfigurable and extensible, and they degrade more gracefully. Also, unlike with single machines that have many processors tightly packed together, in distributed systems energy dissipation and the need for cooling are usually not an issue. However, there are drawbacks, as well. For one, distributed systems are much harder to control and “program” than multiprocessor machines are. Second, they are much more difficult to verify and, in general, formally reason about. Their behavior tends to be much less predictable, detecting various types of errors or failures in a distributed system tends to be much harder than in case of more tightly coupled parallel architectures, and so on.

In case of distributed systems, due to heterogeneity, com-

⁵Our informal definition of a distributed system as a collection of individual computers, that are separated both physically and logically from each other, and then interconnected by some sort of a communication network, is not necessarily the most standard, or most general, notion of a distributed system found in computer science. It, however, suffices for our present purposes, as our notion of a distributed system serves us well as the opposite extreme to the tightly coupled multiprocessor supercomputers discussed in the previous subsection - the opposite extreme, however, only within what we consider herein to be the class of *coarse-grain parallel models*. For more on (classical) distributed systems, see, e.g., [CODK] or [MULL].

munication over distance, networking issues, and other characteristics not found in multiprocessor systems, we can additionally identify, among other, the following major design issues and challenges:

- the nature of the underlying interconnection network (its topology, routing mechanism, etc.);
- the communication model, viz., whether communication is synchronous or asynchronous;
- the resource distribution and sharing mechanisms;
- various other coordination-related issues;
- the privacy and security issues (due to typically having different users at different “nodes”), and mechanisms for meeting the QoS demands in these respects.

It may be appealing to our intuition to distinguish among the parallel and distributed architecture classes of models by employing some terminology from physics as, ultimately, computers are physical systems. The shared-memory multiprocessors are *tightly coupled* parallel machines. An extreme example would be a supercomputer with several thousands of tightly coupled CPUs. The multiprocessor hybrids, such as distributed shared memory architectures, can be viewed as somewhat less tightly coupled. Once the memory is entirely distributed, we have a loosely coupled system. In case of truly distributed systems, where not only there is no shared memory of any sort, but also each processor has its own peripheral devices and OS and is, actually, an autonomous computer, and where a bus within a single machine is replaced with, say, a world-wide network⁶, we get a very loosely coupled distributed computing system.

2. FINE-GRAIN PARALLEL MODELS

Nearly all existing parallel computers built by humans share a number of common features. Among these common features, one central property is that most of the actual parallel computers are constituted of a relatively modest number of relatively powerful processing units. Whether we consider a tightly coupled shared-memory multiprocessor, or a very loosely coupled collection of different computing (sub)systems interconnected via a network, the individual “nodes” where processing takes place are fairly complex, and typically capable of *universal computation* [HOMU]. This, in turn, seems to imply certain practical constraints insofar as *how many* such “nodes” can be “put together” so that they work as a coherent whole, i.e., as a single parallel computer. In case of multiprocessors, the number of different processors is usually in the range of $10^1 - 10^3$. On the other hand, even a global network such as the world-wide web interconnects “only” about $10^6 - 10^8$ computers (as of the early 2000s), and then, making entire such systems (or their sizable subsystems) work in unison is a hard, and often virtually impossible, undertaking. While these numbers of processors may be very impressive in many contexts, they are quite modest in the realms of biology or statistical physics. Is it, then, conceivable to base massively parallel computer models of the future on the principles found outside of computer science and engineering - the principles and paradigms that are readily provided, however, by various complex systems capable of information processing, as often seen in biology or physics?

⁶We take a distributed system over a WAN as the extreme opposite with respect to tightly coupled multiprocessor *supercomputers*.

In this attempt at extracting some common features of the existing parallel computers, we point out the phrase “*built by humans*” at the beginning of the previous paragraph. In case of human-made computers, individual processing units - be it single CPUs possibly along with some local memory, or the entire computer systems that include I/O devices and other “peripheralia” - are, as already pointed out, found to be computationally very powerful, but also fairly complex and costly. Among other issues, in case of tightly coupled computing systems, there are physical and engineering limits as to how many one can hope to ever be able to effectively “pack together”. In case of loosely coupled distributed systems, on the other hand, many problems arise with how to make these different processing elements effectively communicate, coordinate and cooperate, in order to be able to accomplish common tasks efficiently and reliably.

However, these and other common features of man-made parallel computing systems need not be shared by what we may consider as highly parallel information processing systems that “Mother Nature” makes. In fact, in many cases, “Mother Nature” seems to apply exactly the opposite “engineering design principles”. To illustrate this, let’s briefly consider the most sophisticated information processing device (or computer) engineered by Nature that is known to us - namely, *the human brain*.

Unlike tightly coupled parallel computers designed by human brains, the human brains themselves are tightly coupled computers that have (i) a very large number of (ii) highly interconnected (iii) very simple basic information processing units, viz., *neurons*. Any single neuron computes a simple, fixed function of its inputs. On the other hand, there are many neurons in the human brain (around 10^{10} of them), and even more neuron-to-neuron interconnections (some 10^{15} of them), all densely packed in a relatively small volume. The complexity of the human brain - and of the kinds of information processing it is capable of - stems largely (although certainly not exclusively) from these two numbers: a large number of “processors”, and an even larger number of “processor-to-processor communication links”.

There are many other truly fascinating characteristics of the human brains and how they store and process information. While some of these features may be found, at least at the functional level, in man-made computers as well, most of the crucial features of the “computers” that biology builds and the computers that engineers build appear to be rather different from each other. One then may ask, can we learn from Nature, how to build more powerful, more efficient, more robust, less energy wasteful, and cheaper highly parallel computers?

Some ideas for how to approach these challenges from biology-inspired and physics-inspired perspectives have motivated the consideration of a class of abstract “connectionist models” that are the subject of the subsections that follow.

2.1 Artificial Neural Networks

We begin with a brief overview of a class of what can be viewed as “massively parallel models” that are directly inspired by the Mother Nature - more specifically, by biology. These models are based on the idea that nontrivial interconnections and interactions among a very large number of very simplistic “processors” may yield parallel computers that are more scalable, more robust, more energy efficient and possibly even more computationally powerful than the classical parallel models based on extending the

basic sequential von Neumann architecture in various ways. These parallel models whose computational power is based on high interconnectivity and synergetic cooperation among the processing units are often called *connectionist models* in the literature ([GARZ]).

Perhaps the best known and most studied class of *connectionist models* are *artificial neural networks* (**ANNs**). The **ANN** model (or, indeed, a variety of related models that share some important common features) is directly inspired by our knowledge of human brains. Hence, we first briefly review the most important characteristics of the architecture (anatomy) and functionality (physiology) of the human brain.

The human brain is a vast network of *neurons*, together with *receptors* and *effectors* that these neurons interact with [HOFF]. There are several thousands of different kinds of neurons in the brain. Some neurons get their “input” from the receptor sensors and are consequently called *sensor neurons*; others “issue orders” to effectors in charge of the motor control, and hence are called *motor neurons*. The total number of neurons in the human brain is estimated to be $10^{10} - 10^{11}$. Many neurons are connected to tens of thousands of other neurons; these connections are called *synapses*, and the estimated total number of these synapses in the human brain is $10^{14} - 10^{15}$. A single synapse can store only a limited amount of information, about 10 bits, through its coupling strength (or “weight”). Thus, there are about $10^{15} - 10^{16}$ bits of information estimated to be stored in a typical human brain at any time [SEJN].

From the functional perspective, *neurons are the processors of the brain*. However, each neuron viewed as a processor is of a very limited computational power. Namely, any given neuron is only capable of computing a fixed, “hard-wired” function of the inputs that it receives via its synapses and *dendrites* that get signals from other neurons, receptors and/or effectors. Many neurons appropriately weigh their inputs, and then either fire, if the weighted sum has reached or exceeded a specific, hard-wired threshold, or else remain quiescent. Thus, in effect, such neurons compute variants of (weighted) Boolean threshold functions. Many other neurons appropriately nonlinearly *squash* the weighted sum inputs; this squashing enables neurons of nonlinear computation (for more, see, e.g., [HAYK]). Whether a neuron performs any nonlinear processing or not, however, the fact remains that each neuron, viewed as a processor, computes a single, pre-determined function of its inputs. In this respect, a neuron is fairly similar to a *logical gate* as an elementary hardware unit of a typical, silicon-based digital computer.

Insofar as the *computational speed* of a typical human brain is concerned, it has been experimentally established that each synapse is, on average, activated about 10 times per second, thereby yielding an estimated overall brain processing of about $10^{15} - 10^{16}$ bits per second [SEJN].

Billions of neurons in the brain provide the brain with its information processing powers. On the other hand, what exactly constitutes the brain’s *memory*, and how information retrieval from the memory, as well as “writing over” the memory, work in case of the brain, is a nontrivial question. How exactly is the information encoded, stored and retrieved in the brain is a subject of active research in computational brain theory and neuroscience in general. Perhaps the best understood - albeit by no means completely understood quite yet - aspect of neural computation in brains pertains to how neurons encode information. The three rec-

ognized mechanisms are the firing rates, the place/location of particular neurons that fire on a given stimuli, and the population encoding. The issue of how relevant is the *synchrony* of neuron firing is still hotly debated among neuroscientists [ANAS]. While these issues impinge on both the mechanism of neural processing and the nature of memorizing and information storage in the brain, the latter seems to be less understood, and less readily directly translatable into the basic functionality of **ANN** models. We shall discuss below a particular model of “memory” in the context of **ANNs**; however, this is not to claim that the memory in natural neural networks, i.e., brains and central nervous systems, is organized along exactly the same lines as in the case of these artificial neural network models.

How do the brains handle I/O, i.e., how do they interact with the outside world? The receptors continuously monitor both external and internal environments and, in particular, receive various kinds of *stimuli* (such as, e.g., visual or auditory perceptions) from the outside world. When some change in the environment is recorded, receptors send signals to sensor neurons. The effectors, on the other hand, get their input, for example, from the motor neurons, and effect neural endings in the muscles accordingly to produce the desired movement. Thus, from a computational viewpoint, the receptors are *input devices*, whereas effectors play the role of *output devices*. Moreover, there is a certain *layered structure* to human brain - from a functional if not necessarily from an anatomical perspective. To illustrate this layered structure, we briefly outline a very simplified picture of how sensory stimuli (effects) lead to motory movements (responses). Receptors provide input to sensor neurons, which then use their output to feed the input to various kinds of neurons that perform *internal processing*; some of those internal nodes’ outputs then provide inputs to the motor neurons, that then feed effectors with “instructions” for particular motory control actions. This view of functional layers in the human brain has had a major impact on the design of *multi-layered artificial neural networks*⁷

We now turn to main paradigms, characteristics and classes of **ANNs** that are functionally and, in some cases, structurally motivated by our knowledge about the physiology and the anatomy of the human brain. In a typical **ANN**, each “processor” is an imitation of a neuron. These neuron-like nodes are interconnected by a web of *synaptic edges*.

One salient characteristic of the animal and human brains (and nervous systems in general) is that they grow, change and develop over time. This modifiability of nervous systems is called *plasticity* by neuroscientists. In essence, plasticity enables the nervous system, and therefore the living organism, to adapt to various dynamic changes in its surrounding. Two main mechanisms that yield plasticity are conjectured for the human and animal nervous systems, and their variants incorporated into the artificial neural network design. These two mechanisms are:

- (i) *creation of new synapses among the neurons*, and
- (ii) *dynamic modification of the already existing synapses*.

As an example, we consider *Multi-Layer Perceptrons* (**MLPs**) as one of the best-known and most frequently used models of *feed-forward neural networks*⁸. A multi-layer perceptron

⁷Of course, human brains are much more than merely stimulus-response mechanisms such as the ones outlined herein.

⁸Definitions and characteristics of feed-forward and other classes of **ANN** models can be found in any textbook on

is made of an input layer of nodes, followed typically by one or more layers of the so-called *hidden nodes*, which are then followed by a single output node, or a single layer made of several output nodes. The input layer is iteratively “fed” various inputs, typically *patterns* that need to be recognized by the network, or some specific properties of these patterns learnt. The main learning mechanism in this kind of networks is that, as the computation is iterated, the weights of the synaptic edges keep dynamically and interactively changing.

To summarize, when viewed as an adaptive “learning machine”, an **ANN**, such as a *MLP* feed-forward network, functions, in essence, as follows:

- (i) it acquires knowledge through an iterative learning process,
- (ii) it “stores” the knowledge already accumulated in the current (but modifiable) values of synaptic weights, and
- (iii) it may modify and/or increase its knowledge about the environment (presented to the network in the form of the input patterns) via dynamic modification of these synaptic edge weights.

Let us now turn to the important issue of *memory* in neural networks. As there is no separate physical “memory storage” in a neural network, the information is stored solely in the nodes and the edges of the network. The “program” is distributed across the nodes: each node stores the information, what function of its inputs it is supposed to compute. A distinguishing characteristic of artificial neural networks is that the “data”, i.e., the accumulated knowledge and information are dynamically stored in edges as real-valued and modifiable *edge weights*. The initial edge weights represent “initial state” of the network; these initial weights are either some particular pre-specified values, or (more commonly) randomly generated real values from a certain pre-specified range. The edge weights then dynamically change as the computation proceeds from one iteration to the next, as the network keeps receiving and processing the incoming patterns (i.e., new inputs). In a typical **ANN** application, it is also said that the network *learns* about or from the input data, via this continued dynamic changes and adjustments of the edge weights. Thus, in case of **ANNs**, it is the communication links (synapses) and the numerical values that they hold and dynamically change (synaptic weights) that take the role of the input / output / working tape of a Turing machine, or, equivalently, of the storage medium (“memory”) of any standard digital electronic computer.

After having discussed processors, memory and basic functionality of **ANNs**, we outline some of the main classes of neural network models. The simplest model, and the one that is usually considered to be the foundation stone of the whole area of **ANNs**, originally introduced in 1942 by McCullough and Pitts ([MCPI]), is the *linear perceptron*. Linear perceptron is a simple mathematical model of a single neuron that computes a *Boolean linear threshold function* of its inputs. Therefore, all a single perceptron can do is classify each input into one of the two linearly separated classes. However, by appropriately combining a few perceptrons, one can simulate the usual Boolean operations such as *NOT*, *OR* and *AND*. Therefore, networks made of several layers of multiple linear perceptrons can simulate arbitrary logical and integer-arithmetic operations of digital comput-

the subject; one recommended reading is [HAYK]. We very briefly review some of the most important classes of **ANNs** at the end of this subsection.

ers. One can generalize the linear perceptron to obtain models of artificial neurons that, instead of a linear, can compute, e.g., a quadratic, or, more generally, an arbitrary polynomial *separator function*. A more important generalization, however, is the already mentioned *Multi-Layer Perceptron (MLP)*, perhaps the single most important class of *feed-forward networks*.

Feed-forward networks are **ANNs** whose topologies contain no loops. That is, all the node layers are totally ordered, and the signals always propagate from the layer closer to the input to the layers (typically, but not necessarily, to the very next layer), that are closer to the output; in particular, no signal ever propagates backwards.

If loops are allowed, that is, if signals can travel from a given layer to one of the previous layers, then there is a *feedback* in the network. Of particular importance are the so-called *Recurrent Neural Networks*, where some or all of the output values computed by the output layer nodes are subsequently fed back to some of the nodes in the layers preceding the output layer. In some architectures, outputs from the output nodes are fed back to the input nodes, while in others, some of the hidden layer nodes receive their input both from the input and/or the earlier hidden nodes (feed-forward signals), and from the output nodes (feedback signals). For more, see, e.g., [HAYK].

There are many ways of classifying various types of **ANNs**. Some critical design parameters that lead to different types of networks are the number of layers, and the kind of *squashing functions* that the nodes that do nonlinear processing use. In terms of the direction of signal propagation at the functional level, as well as the underlying network topologies at the architectural level, we distinguish between the loop-less feed-forward networks, and the feed-back networks with loops for back propagation of information. In terms of the learning paradigm, networks designed for supervised learning tend to be considerably different from the networks for reinforcement learning or those for unsupervised learning. For instance, *MLP* is a standard model for supervised learning problems whereas *self-organizing Kohonen’s maps* (see, e.g., [HAYK]), on the other hand, are exemplifying the unsupervised neural learning paradigm.

Some important **ANN** classes include:

- *Radial Basis Function* neural networks;
- *Hopfield Networks* that are based on the auto-associative memory paradigm;
- *Adaptive Resonance Theory (ART)* networks and their various extensions;
- topological self-organizing maps, such as *Kohonen maps*, used in unsupervised learning;
- unsupervised learning models based on *reaction-diffusion*, etc.

Let us summarize in what most crucial respects are the connectionist models fundamentally different from other parallel models. An artificial neural network, as a typical connectionist massively parallel computational model, has a very large number of very simple processing units. The hierarchy of how these processing units (“neurons”) are organized tends to be fairly simple, with a few layers so that each layer has a distinct function. The only “memory” for the data are the modifiable edge weights. The capability of potentially highly complex computation stems from the following three critical factors:

- (i) the large number of processing units;
- (ii) an even larger number of processor-to-processor con-

nctions (or “*neural synapses*” in the lingo of biology); and

(iii) the capability to simulate “writing over the storage medium” via dynamically changing the edge weights (as the network’s only “RAM memory”).

ANNs are a rich class of models and an active research area for several decades now. **ANNs** have been studied by researchers coming from fields as diverse as biology, neuroscience, computational brain theory, artificial intelligence, and “hard core” computer science. Therefore, literature on this field is extremely rich and diverse⁹. Our focus herein is not on relevance of **ANNs** in, say, neuroscience or artificial intelligence, but on what paradigms and promises neural nets and other connectionist models have to offer to the theory and practice of parallel computation. **CA**, another class of massively parallel models, are discussed next.

2.2 Cellular Automata

We next consider another model of massive parallelism (or, rather, another class of such models), that is not directly inspired by biology, neurons and the brain like **ANNs** are, but, instead, is chiefly inspired by physics. This class of models are *Cellular Automata* (**CA**). Together with **ANNs** and *Random Networks* ([GARZ]), **CA** are also referred to as *the models of fine-grain parallelism*, in contrast to the *coarse-grained parallelism* of multiprocessors and networked distributed systems. In this subsection, we also relate **CA** to **ANNs**, and then, in the next section, qualitatively compare and contrast both these two classes of connectionist fine-grain parallel models with the classical, coarse-grain parallel and distributed computing models.

CA were originally introduced in [NEU2]. One approach to motivating **CA** is to first consider *Finite State Machines* (**FSMs**) such as *deterministic finite automata* (**DFA**); for a comprehensive introduction to **DFA** and their properties, see, e.g., [HOMU].

Let us consider many such **FSMs**, all identical to one another, that are lined up and interconnected together in some regular fashion, e.g. on a straight line or a regular 2-D grid, so that each single “node” is connected to its immediate neighbors. Let’s also eliminate any external sources of input streams to the individual machines that are the nodes in our grid, and let the current values of any given node’s neighbors be that node’s only “input data”. If we then specify the set of the values held in each node (typically, this set is $\{0, 1\}$), and we also identify this set of values with the set of the node’s *internal states*, we arrive at an informal definition of a cellular automaton. To summarize, a *classical CA* is a finite or infinite regular grid in one-, two- or higher-dimensional space (or, more generally, another finite or infinite regular undirected graph), where each node in the grid is a particularly simple **FSM**, with only two possible internal states, the quiescent state 0 and the active state 1, and whose input data at each time step are the corresponding current internal states of its neighbors. All the nodes execute the **FSM** computation in unison, i.e. (logically) simultaneously. Thus, **CA** can be viewed as a parallel model where *perfect synchrony* is assumed. For different variants and generalizations of the basic **CA** model, see [GARZ].

Another way of informally introducing **CA** is to start from **ANNs**, and make some modifications in their definition.

⁹As reading suggestions, we strongly recommend, in addition to [HAYK], [HOFF] and [GARZ] as examples of very good computer science texts, a highly acclaimed book by two neural scientists, [CHSE].

First, individual “cells” or “nodes” in a **CA**, viewed from a neural network perspective, rather than computing discrete-valued or real-valued functions of analog (real-valued) inputs, should instead compute discrete-valued functions of discrete input values, i.e., the Boolean functions of a kind that finite state machines are capable of computing. Second, all the **CA** “cells” or nodes are identical to one another, i.e., all cells simulate the same **FSM**. Third, edges carry no weights. Fourth, the underlying graph structure, rather than being layered as in classical **ANNs**, is highly regular. In particular, a **CA** locally looks the same everywhere: the structure of a neighborhood of any particular node is identical to the neighborhood structure of any other node.

The two most studied classes of **CA** are the 1D and 2D *infinite* cellular automata (see [WOLF] or [GARZ]). A 1D (infinite) **CA** is a countably infinite set of *nodes* (also called *sites*) connected by an infinite straight line. The geometric neighborhood of each node - viz., its left and right neighbors¹⁰, need not coincide with the *functional neighborhood*. For example, the next state of a node may depend on the two nodes immediately to the left, and two nodes immediately to the right from it, in which case the functional neighborhood is of size 4 rather than 2. Similarly, 2D infinite **CA** are defined over a two-dimensional regular planar grid as the underlying *cellular space*; this grid is isomorphic to the set of points in 2D Euclidean plane with both integer coordinates. While **CA** can be defined over other *regular graphs* (either finite or infinite), the 1D (“line”) and 2D (“plane grid”) **CA** are the best known and most studied special cases.

It has been shown that infinite **CA** can be designed so that they are capable of universal computation (that is, can effectively simulate a universal Turing machine) - even in the 1D case (see [GARZ] and references therein). Hence, **CA** appear to be a legitimate model of a *universal (massively parallel) computer*¹¹.

We now mention some conceptual problems with the **CA** model, in terms of how (un)realistic it is. In classical **CA**, all the nodes (i.e., the individual **FSMs** at those nodes), when they update their states “in parallel”, are really assumed to compute in perfect synchrony. This is hard to justify, especially in the infinite automata case. Except for some spooky quantum mechanical effects¹², there is no (instantaneous) action at a distance - and certainly not at an arbitrarily large distance. One line of defense could be that it is not the actual *physical simultaneity* of the node updates that is required of a **CA**, but rather merely *logical simultaneity*. As long as the nodes are functioning faultlessly and with a *known* finite bound on possible communication delays, it may be argued that this assumption is not so far-fetched (at least not conceptually), given that the “programs” these nodes execute are computations of rather simple functions.

Another important issue is whether allowing “arbitrary initial configurations” in case of infinite **CA** should be admissible, even at an abstract level. First of all, there are uncountably many such configurations (i.e., infinite strings of 0s and 1s), while there are only countably many possible computer programs. Second, let us consider classical Tur-

¹⁰Assuming the nodes are successively enumerated by integers, the immediate neighbors of node n are its predecessor and its successor, i.e., nodes $n - 1$ and $n + 1$.

¹¹**CA** are capable of even more, i.e., of computing functions that are not Turing-computable, if infinite **CA** with non-finitary starting configurations are allowed [GARZ].

¹²An interested reader is directed to quantum physics literature on the *Einstein-Podolsky-Rosen paradox*.

ing machines with their infinite memories. We recall that a (*universal*) *Turing machine* (**TM**) is a canonical abstract model of a general purpose (sequential) digital computer (e.g., [PAPA], [HOMU]). While the memory tape of a **TM** is assumed infinite, only finite starting configurations (i.e., “inputs”, that in general may include both the program and the data) are allowed. This also ensures that, at any step of the computation, only finitely many “memory cells” will have been accessed and/or written, with the rest of the cells remaining “blank” or, in the lingo of **CA**, “quiescent”. The justification for allowing only finite inputs in case of Turing machines (and computer programs in general) is evident. The finite input restriction can be translated to infinite-sized **CA**, as well - by requiring that the only permissible input configurations are the “finitely supported configurations” where all but finitely many cells are initially required to be in the “*quiescent state*” (that is, the 0 state). Given the nature of coupling and therefore information propagation in **CA**, this requirement, in particular, ensures the finite speed of information propagation¹³. This, in turn, implies the following:

- (i) there are only countably many starting configurations, countably many reachable configurations after any finite number of steps, and countably many **CA** “programs”;
- (ii) after any finite number of parallel steps of any **CA** starting from any finitely supported initial configuration, there will be only finitely many non-quiescent states; and
- (iii) there is no spooky action at unbounded distances.

For these reasons, we argue that, if one wants to consider infinite **CA** as a legitimate massively parallel model of computation, one should confine his study to the finitary configurations only. Beside the finitely supported configurations, one would also wish to allow the *space-periodic configurations*, such as, e.g., $(10)^\omega$, as they have finite descriptions and finitary dynamics¹⁴.

We now turn to a brief comparison and contrast between **ANNs** and **CA**. We have already pointed out the main complexity tradeoffs in model specification as one moves from **ANNs** towards **CA** (or vice-versa). We have also emphasized that it is precisely the dynamically modifiable *edge weights* in **ANNs** that make neural networks a flexible computational model capable of adjusting to and learning from its input data. On the other hand, there is no learning in **CA**, at least not in the usual sense of the word¹⁵. Moreover, the only “data” each node computes on are the states of that node’s neighbors (possibly including the node’s own state). Next, the power of a neural network is typically to a large extent due to the fact that each “processor” tends to interact with quite a few other processors; that is, **ANNs** tend to have fairly complex and dense interconnection topologies. Also, many neural networks are locally fairly heterogeneous in a sense that the pattern of node-to-node connections may significantly vary from one node to the next, depending on the node’s layer and its overall role in the network. On the other hand, locally, a **CA** by definition looks the same

¹³For example, in the simplest, 1D case, the change of state of a cell $[i]$, after one (parallel) step of the cellular automaton’s evolution, can have effect only on cells $[i-r], [i-r+1], \dots, [i+r-1], [i+r]$, where r is the radius of the coupling.

¹⁴Computations of a given **CA** with a finite interaction radius r on a given spatially periodic input such as $(10)^\omega$ can be simulated by an appropriate *finite cellular automata with circular boundary conditions*.

¹⁵We don’t consider herein the phenomena such as, e.g., self-organization as a “proper” kind of learning.

everywhere, in that a neighborhood of each node looks exactly like the neighborhood of any other node. Also, while the actual functional neighborhood of a **CA** node need not coincide with its physical neighborhood (with respect to the chosen metric or distance function), typically the functional neighborhood sizes are still fairly small. That is, each node is functionally dependent on only a handful of other, nearby nodes - and the pattern of this dependency is the same across all the nodes.

We conclude with a note on the origin of **ANNs** and **CA**. These two classes of fine-grained parallel models were originally designed with different primary applications in mind, and hence these fundamental differences in their respective architectures. A succinct - and therefore necessarily oversimplified - distinction between the typical “target applications” for **ANNs** and those for **CA**, is that **ANNs** are most commonly employed in computational tasks that involve some form of *learning*, whereas **CA** find most of their applications in studying various paradigms of complex systems’ *dynamics*. Both classes of models, however, have been also used in a variety of other contexts, including massively parallel general purpose computing.

3. FINE-GRAIN VS. COARSE-GRAIN PARALLEL MODELS

Our next goal is to outline a quantitative and qualitative comparison and contrast between the fine-grained connectionist parallel models, such as **ANNs** and **CA** on one, and models of coarse-grained massive parallelism, such as multiprocessor supercomputers or the usual networked distributed systems, on the other hand. We first compare the underlying architectures, or physical structures, of these two distinct classes of parallel models in subsection §§3.1. Functional differences between fine-grained and coarse-grained parallel models will be the subject of §§3.2.

3.1 Architectural comparison and contrast

The first striking difference between the two classes of parallel models compared and contrasted in this paper is with respect to the *number of processing units*. In case of multiprocessor machines the typical range (as of the early first decade of the 21st century) is from a few, to a few dozen, up to a couple of hundred processors. The most complex, powerful and costly multiprocessor machines (the “supercomputers” of the 1990s and the early first decade of the new millennium) may have from a few hundred up to several thousands of processors. Some examples include *Fujitsu Parallel Server AP3000 U300* with 1,024 UltraSPARC scalar processors, and *Hitachi SR2201* with 2,048 pseudo-vector RISC processors.

If, instead of individual multiprocessor machines, one considers a networked distributed system as a single parallel computer that just happens to be physically distributed, then the number of “processing units” is a single such system can surpass the corresponding numbers in multiprocessor supercomputers by 3-4 orders of magnitude. Of course, making systems made of millions of computers that may be dispersed around the globe, have different users with different resources and goals, may have different architectures, operating systems, etc. - all work in unison, so that such a collection of networked computers can be reasonably considered a single computing system, is a daunting, often literally impossible, engineering task to undertake.

On the other hand, “computers” with 10^{10} neurons (“processing units”), and 10^{15} synapses (“communication links”) between them, as already noticed, are readily provided by Mother Nature. Moreover, building (artificial) neural networks made of say millions of (artificial) neurons may not be as impractical as it may seem at first glance. Actually, it is entirely possible that the basic building blocks of massive neural networks (beside human or animal brains) are already readily available - we just need to figure out where to look for them.

An example of a “massively parallel” physical system, where interactions between “computing elements” are taking place at different scales, and which hold some promise for massively parallel computing, are the *solitons*. More generally, different kinds of *excitable, nonlinear physical media* where the phenomenon of *reaction-diffusion* can be utilized for cheap and physically readily realizable (at least in principle) massively parallel computation, and different types of reaction-diffusion processes that are promising in terms of their information-processing potentials, have been extensively studied. Most of this exciting work, however, is done by physicists and experts in complex dynamical systems, rather than computer scientists. For more on this subject, see, e.g., [ADAM].

Another, more exotic model of massive fine-grain parallelism is provided by the *quantum computing* paradigm, esp. in the context of *Nuclear Magnetic Resonance* (NMR) spectroscopy, where the underlying systems have 10^{20} or more “basic processing units” (typically, magnetic spins of certain kinds of atoms or molecules); see, e.g., [CORY]. Thus, fine-grain parallel computers, with ten or more orders of magnitude more “processors” than the coarse-grain parallel computers as we know them, are in a sense already provided by the Mother Nature. Our task, then, is to find them, recognize them when we encounter them, and learn how to utilize their information-processing potentials - especially in terms of controlling (or “programming”) the underlying physical or biological systems and their dynamics.

There are other important distinctions between typical processing units in massively parallel connectionist fine-grain models, and those in coarse-grain multiprocessors or distributed systems. The main difference between individual processors in the two classes of models is that, whereas a single processor in a coarse-grain parallel computer is powerful, expensive, and typically tends to dissipate a good deal of energy, a processing unit in a **CA**-based or a neural network based computer would be expected to be very simple, very cheap, and highly energy efficient. Indeed, a single “node” of a networked distributed system is, actually, a complete computer by itself, with one or more CPUs, its own memory, its own connection and communication links to the rest of the distributed system, and possibly its own I/O and other peripheral devices. In case of multiprocessors, each node is at the very least a separate CPU, possibly with some of its own local memory (say, local cache). In any case, each computing node in any coarse-grain parallel model is of complex structure, and capable of complex computation. In contrast, a single node of a fine-grain connectionist model only computes a single pre-defined (and typically very simple) function.

Another major difference between fine-grain parallel models on one, and coarse-grain parallel computers on the other hand, is the nature of memory or the read/write/storage medium. We have already discussed what in essence constitutes the storage medium in **ANNs**.

Since the edge weights can be, in principle, arbitrary real numbers, meaning that an **ANN** can presumably store the actual real numbers (with an infinite precision) - something that no digital computer can accomplish - in this sense neural networks can be viewed as an *analog* (as opposed to *digital*) computer model. However, the outputs of **ANN** computations are either discrete or readily discretizable. If we simulate an **ANN** on a digital computer (which is how study of neural networks is usually done), the input values are not actual real numbers, but rather their rational, finite-precision approximations. Furthermore, even if one manages to build an actual artificial neural network with analog edge weight “inputs”, the nodes that receive these real-valued inputs will, ultimately, have their sensors or other measuring devices (that “read” the real-valued inputs) that only operate with a finite precision, thereby, ultimately, rendering such a neural network digital, after all. Thus, **ANNs** can still be considered discrete and, therefore, digital (or readily “digitizable”) computer models - albeit with a “memory” much different from the storage media of the electronic digital computers as we know them.

In case of **CA**, the contradistinction of the nature of “memory”, “storage” and “input” with respect to the corresponding notions in electronic digital computers is even more striking. Consider a single “processor” of a cellular automaton. The entire input this processor gets are the current states of its neighbors. Assuming the domain is binary, each processor is a simple finite state machine that can be in one of the two states only, 0 or 1, and that may or may not change its state, this decision being solely based on the states of its neighbors as the only “input”. Insofar as memory storage is concerned, each node of such automaton with a $\{0, 1\}$ alphabet has exactly one bit of memory - its own current state¹⁶. Since all nodes of a **CA** compute the same function (i.e., simulate one and the same finite state machine), one may argue that **CA** are as simplistic as it gets when it comes to (i) processors, (ii) memory, and (iii) interaction with the outside world (or what a computer scientist would call “I/O”). Thus the power of the **CA** model stems entirely from *interactions* among the components (nodes) of the system itself, and the *synergy* among these individual nodes and their local computations.

The first two major qualitative and quantitative differences between the coarse-grain and the fine-grain parallel models, viz., the differences regarding the nature of processing units and the nature of memory/storage, are highlighted from a traditional, “von-Neumannian” viewpoint. In any variant on the theme of the classical von Neumann computer architecture, whether with one or with many processors, an *a priori* tacit assumption is always made, that there is a clear distinction between processors and memory.

In the classical von Neumann model, there is a processor (or several processors), a memory both physically and logically separated from the processor(s), and some sort of the interconnection link (data path) between the two. In contrast, the distinction between “processors” and “memory” in case of **ANNs** is more subtle, as the only “memory” that stores the *programs* of a neural network are actually the processors themselves, whereas the only “memory” for storing

¹⁶One may also consider *memoryless CA*, where the next state of a node or site in the cellular space does not depend on its own current state, but only on the states of the nodes belonging to some *proper* pre-specified neighborhood of that node.

the data are the links (synapses) connecting the processors, i.e., the processor-to-processor (and possibly “processor-to-outside-world”) interconnections. Moreover, in case of **CA**, it can be readily argued that there is virtually no distinction between “processors” and “memory”, as each “node” is a “processor”, but its state is also “data” or “input” that is stored in the node for its neighbors to use and compute with. We also observe that, together with the node’s coordinate(s) or “ID”, the node’s *state* is the only distinguished property any node has that matters for a **CA** computation. The node’s state is also *the only* property of a node (or, more precisely, of the **FSM** that each node of the automaton is computationally equivalent to) that changes dynamically, and that can affect (or be affected by) the subsequent changes of the states of near-by nodes. Thus, there is virtually no “storage” or “I/O” or “read/write medium” in **CA** beside their processing units.

Now that we have identified some fundamental “hardware” differences between the coarse-grain and the fine-grain abstract models of parallel computers, we focus next on the intrinsic differences between the two classes of computing models when it comes to how these computing machines work - that is, how is information processed and how is computing performed in each case.

3.2 Functional comparison and contrast

Given their entirely different architectures from the classical parallel or distributed computing systems, it is not surprising that the fine-grain connectionist parallel models compute in a radically different way from their classical, coarse-grain counterparts. To emphasize the most profound differences, we first briefly review how classical computers process information.

In the von Neumann model, the main parts of the system are one or several processors, a storage medium (memory), and some links connecting processor(s) with memory. A processor fetches data from memory, processes that data according to a pre-specified set of rules (given by the computer program), and then stores the (possibly modified) data back to the storage. We argue that this model of information processing largely stems from a Turing machine-based view of a digital computer and its computing process.

A Turing machine (**TM**) consists of a control unit, that can be in one of finitely many states, an infinite tape (or “memory”) where the input is received, processing of the data performed by reading from and writing to the tape, and the output (if any) written, and a head or “cursor” that points to that cell (location) of the tape (memory) where the next computation step is to be carried out. The tape is not merely a storage medium, but also the “working space” where computation is taking place. In particular, the ability of a **TM** to *write over the current content of the tape* in general, and to overwrite its input in particular, is critical for the computational power of the model. Without the ability to write the tape, a **TM** would be no more powerful than a **DFA**. Unbounded memory, together with a possibility of *random memory access*, is also crucial and necessary for Turing machine’s ability of universal computation. [SIPS] summarizes the fundamental differences between **TMs** and **FSMs** by noting these four characteristics of Turing machines: (i) being able to both read from and write to the tape (that is, memory), (ii) the read/write head can move both to the left and to the right (thereby assuring the capability of *random memory access*), (iii) the tape is infinite

(since no *a priori* imposed bound on the memory size will necessarily suffice for computation of an arbitrary **TM** on an arbitrary input), and (iv) the special states for acceptance and rejection take immediate effect, unlike in the case of **FSMs** such as **DFA**. These properties of Turing machines, and particularly their features (i) and (ii), translate fairly directly into the actual physical operations of digital computers as we know them.

Reading from and writing to the storage medium, as we already pointed out, is at the very core of the functioning of digital computers. If the tape has the role of the memory, then the head pointing to a location or cell on the tape is analogous to the data from some particular memory location being chosen to be operated upon (and therefore fetched, i.e., loaded from memory to a processor). Left and right head movements along the tape enable the **TM** to access all memory locations, and, in particular, allow for the *random memory access*. This particular capability is crucial for the computational power of the **TM** model, in contrast with those computational models with restricted kinds of memory access, such as the the *push-down automata* (**PDA**) with their stacks - as such restricted accesses to memory do not empower the underlying formal automata models with the capability of universal computation¹⁷.

In contrast, it is immediate that **ANNs** and **CA** process information and compute in a fundamentally different way from the above outline of Turing machines’ (and therefore digital computers’) way of computing. In these fine-grain connectionist parallel models, there is no “tape” or “storage medium” to write over. Consequently, both processing and storing information is done in an altogether different manner from how virtually all the existing (electronic) digital computers, implementation details aside, process and store data.

The profound differences in the architectures between the fine-grain and the coarse-grain parallel models - the former do not follow the processors-separated-from-writable-storage paradigm - also imply some major functional differences. The fact that there is no writing over the storage medium in the connectionist models is perhaps the most striking functional distinction, but not the only one.

Another fundamental functional difference between fine-grain and coarse-grain models is how they interact with the outer world. This difference is most striking when **CA** are compared with any model of classical digital computers. Namely, **CA** are *closed physical systems*. The only “input” a **CA** ever uses is its own *global configuration*. **CA** are neither affected, nor can they affect, their environment. However, it should also be noted that an infinite **CA** has an uncountable number of possible configurations (and therefore possible inputs), whereas any classical model is assumed to receive as inputs only finite strings defined over an appropriate finite alphabet, and therefore any such model, from an abstract **TM** to an actual electronic digital computer where we assume unbounded memory and “abstract away” the issue of finite range of storeable/representable machine numbers¹⁸, can have only countably many different inputs.

¹⁷ Namely, **PDA** can recognize only a proper subset of all decidable languages of finite strings, namely, the *context-free languages*; for more, a reader not familiar with the fundamentals of the formal language theory is referred to [SIPS] or [HOMU].

¹⁸ ...but we *do not* “abstract away” the fundamental property that each number can be represented with *perhaps an arbitrary, but still only finite precision*.

ANNs, on the other hand, are more interactive with the outside world than **CA** are: for example, one can pick and choose “from the outside” what patterns to “feed in” to the input nodes of an **ANN**. However, only a part of the **ANN**’s “state” is observable from the outside - namely, given an input, which of the output layer neurons fire, and which do not. It is not coincidental that the nodes where the internal processing is taking place, that is, the nodes “in between” the input and the output nodes, are called *hidden* in the **ANN** literature. This name captures the idea that this part of the network’s state is not observable from the outside. Moreover, to fully describe the state of the network at any time step, one also needs to know all the current synaptic edge weights - another set of parameters that are, in a sense, inherently *internal* to the system, rather than observable from the outside. In contrast, **CA** are entirely transparent: to fully describe the state of a cellular automaton, in addition to knowing its underlying topology and the **DFA** that all the nodes simulate, one “only” needs to know the current state of each of the automaton’s nodes. Assuming the state of an infinite **CA** does not have a finite support or some characteristic (such as spatial periodicity) that allows for a finitary specification, however, the information content of “only knowing the states of all nodes” is infinite. On the other hand, to fully and perfectly accurately capture the state of an **ANN**, one needs to know real-valued edge weights with a perfect (and, therefore, in general, *infinite*) precision - another infinite information content. In many contexts, this may be unacceptable, as briefly discussed in §2 (and in much more detail in, e.g., [GARZ]).

Back to what input and output and, more generally, interaction with the environment look like in case of **ANNs** and **CA**, one may argue that, at the very fundamental level, the “I/O” of the connectionist models is rather different from what we see in digital computers. Furthermore, **ANNs** and **CA** are also profoundly different from one another when it comes to how they, viewed as physical systems, interact with their environments, and how much and what kind of information an outside observer needs to know about them in order to be able to fully specify their respective “states”.

We have discussed the main *functional* respects in which **ANNs** and **CA** are fundamentally distinct from the classical digital computers. As in case of the structural comparison and contrast in the previous section, the focus was on the fundamental conceptual differences, rather than any particular engineering design or implementation idiosyncrasies. At the very core level, we have shown that the connectionist models are rather different in how they receive, process and store information from their classical (electronic) digital computer counterparts.

3.3 Some comparative advantages of the connectionist fine-grain models

After summarizing both architectural and functional differences between the connectionist models on one, and the classical models underlying electronic digital computers, on the other hand, we now briefly reflect on some (expected) comparative advantages of each. One of the goals of this work as a whole, and the primary purpose of this particular section, is to motivate a discussion on the potential promises of designing and building actual massively parallel computers that would be largely based on the connectionist paradigms.

We find it reasonable to assume that, for comparable numbers of neurons (or neuron-like switching elements) and synapses, an actual connectionist, **ANN**-like computer would not be able to compute either much more or much better than what human brains can compute¹⁹. That is, the power of a reasonably developed and functional human brain should be an (asymptotic) upper bound in terms of computational power and efficiency on any connectionist artificial computer with roughly 10 - 100 billion neuron-like processors, and 1,000 - 10,000 as many processor-to-processor connection links. Now, if digital electronic computers outperform human brains in case of certain computational tasks, then it is to be expected that **ANNs** of sizes comparable to the typical size of a human brain will also be outperformed by the same (or similar) digital electronic computers on those same computational tasks, as well. One example is multiplying very large integers or floating-point “reals”. Another example is searching through very large databases of structure-free data (“*unordered databases*”). There are certainly many other such application domains where, assuming a connectionist parallel computer has available “hardware” comparable to the “hardware” we humans are endowed with, such a computer should not be realistically expected to solve typical problems from those domains faster than, or even comparably as fast as, the digital electronic computers of the early 21st century.

On the other hand, there are plenty of application domains where humans perform very well, with quickness and ease, whereas the fastest computers running the best (known) algorithms struggle, take a very long time or, sometimes, utterly fail. *Pattern recognition*, where the patterns are highly structured and/or context-sensitive, is one such example: no computer or artificial “expert system”, for example, can get anywhere close to the speed and accuracy with which humans recognize and distinguish between, e.g., different human faces or other similar context-sensitive, highly structured visual images²⁰.

The greatest computational advantages of human brains over the existing “silicon brains” are usually encountered where it is the organic brains’ *plasticity* that make us excel with respect to the silicon (or other “non-organic”, man-made) computers. The computational tasks where neural plasticity easily beats silicon, therefore, should be expected to have considerable *learning*, *dynamic interaction with the environment*, and/or *adaptability* components embedded in them, and where quantities to be computed on are intrinsically *fuzzy* rather than *crisp*.

One can multiply two 100-digit numbers without any learning or any interaction with the environment (other than reading in the input), but by merely following a simple, fixed-size-and-complexity set of arithmetic rules. That is why digital computers are so much better than humans when it comes to multiplying large numbers. On the other hand,

¹⁹We deliberately stay away altogether from any discussion of those functions of human brains for which it may be debatable whether they are *computational*, or, indeed, from any debate whether such functions of human brains exist at all. Likewise, we stay away from questions such as whether *consciousness* and *mind* and *soul* are real, computable, solely due to information processing in the brains, and the like.

²⁰We remark that correctly classifying a set of visual images is certainly a computational task. Any conscious experience a human may have while performing such a task, on the other hand, can be readily argued not to possess this, purely computational quality.

those problem domains where the computing agent has an on-going, dynamic interaction with complex yet structured environments, and where computations may have an intrinsically uncertain or fuzzy component to them, is where humans still excel - and likely will continue to excel in the foreseeable future.

After this brief overview of why, when it comes to the problem solving speed and effectiveness, each of the connectionist and non-connectionist models is to be expected to perform much better than the other in case of certain problem domains that happen to be highly “compatible” with the reigning model’s structure and “computing style”, we return to the central theme of this subsection. In making comparisons and contrasts between the fine-grain, connectionist-based parallel models on one, and coarse-grain parallel models on the other hand, we would like to identify some of the most important comparative advantages that can be reasonably expected of the hypothetical (yet to be) physically realized parallel computers that are designed along the general lines of **ANNs** or other fine-grain parallel models. In this pursuit for some such generic potential advantages of the connectionist models, we would like to identify some quantifiable metrics where the actual massively parallel computers designed along the connectionist paradigms (once and if they are actually built) should be reasonably expected to outperform their classical, multiprocessor or distributed system counterparts - even if we assume (much?) faster CPU cycles, memories, and communication networks in case of the latter, but within the well-understood limits imposed by the laws of physics (as discussed in §1). The claims here-with are necessarily only tentative, as the actual computer hardware along the connectionist lines is yet to be built and deployed.

The most significant possible advantages of the actual fine-grain parallel computers designed according to the basic connectionist assumptions, in our view, would be expected to be the following:

- *scalability* in terms of keeping increasing the number of processing units and effectively getting more exploitable parallelism for the additional hardware;
- *avoiding the slow storage bottleneck*: the storage media (“memories”) being much slower than the processors are not an issue in **ANN**-like or **CA**-like models;
- *flexibility, adaptability* and *modifiability* of such a system, without requiring to re-wire or explicitly re-program the entire system or its major components;
- *robustness* and *reliability* in the presence of noise;
- *graceful degradation*: connectionist computers should be expected to degrade much more gracefully in presence of either node or communication link failures than their classical, non-connectionist counterparts;
- *energy consumption*: connectionist parallel computers, assuming sufficient similarity to brains, should dissipate much less heat than classical parallel or distributed computers of a comparable computational power.

We shall try to very briefly justify some of these conjectured comparative advantages of the fine-grain parallel models by briefly comparing what is already known (when it comes to the quality metrics above) with the corresponding features in classical parallel multiprocessors and networked distributed systems.

First, we consider some advantages of **ANN**-like or **CA**-like models with respect to the usual tightly coupled parallel computer architectures. “Packing” several hundred or sev-

eral thousand processors, for example, in a new Cray supercomputer is a major challenge. Yet “packing” 10^{10} neurons with all their interconnections inside a fairly small volume (a human skull) has been very efficiently solved by Nature. Regarding the (relative) slowness of storage access in case of any von Neumannian computer architecture, we observe that, while both processors and memories of digital electronic computers keep getting faster, these growths in speed are rather disproportional. The architectural and hardware developments over the past 30-40 years have been such that the gap between CPU speeds and memory speeds has kept growing - and is expected to keep growing in the foreseeable future. This problem, however, should not be expected at all in connectionist-based parallel computers, due to an entirely different paradigm of information storage and retrieval.

Another important advantage of the connectionist models is that, for example, the aforementioned 10^{10} brain neurons dissipate relatively little energy and, unlike, say, the Cray supercomputers, do not require major engineering and financial burdens related to cooling²¹. Some quantitative comparisons are in order in the context of heat dissipation. Perhaps the first such evaluation and comparison between the basic electronic and neural switching elements was done by John von Neumann in the 1950s: he compared the leading technology of that time, viz., a vacuum tube, with a typical human brain neuron. The neurons considerably outperformed the vacuum tubes with respect to (higher) interconnectivity, (higher) density, (smaller) size, and (lower) energy consumption [NEU1]. Three decades later, a much more advanced technology, namely, the MOSFETs, were compared with the brain neurons. It turns out that this leading silicon switching element of the 1980s outperforms the neuron by *six orders of magnitude* in terms of the execution speed, while having similar size, packing density and failure rates to those of brain neurons. The two important metrics where the neurons still remain far superior with respect to the silicon (in this case, MOSFETs) are the interconnectivity and the heat dissipation [AKFG]. The authors of that study conclude that “Nature has found an architecture [that is, the brain] which uses highly interconnected, slow devices to make extremely fast sensory processing systems” ([AKFG]). Indeed, processing of the sensory stimuli is one important - but by no means *the only* - class of computational tasks where the neural “wetware” still considerably outperforms the man-made (semi-conductor or any other) hardware.

In addition to thus far unchallenged reign of the neural wetware when it comes to interconnectivity and energy dissipation, the tightly coupled connectionist massively parallel computers that would be based on such neural “architectures” can be also expected to exhibit a very graceful degradation. Namely, neurons in our brains keep dying throughout our lives, yet, for at least a few decades, we seem to be functioning reasonably well in spite of that. A similar phenomenon applies to dying out of neural synapses, as well. In contrast, the tightly coupled computer or other engineering systems we build tend to easily and completely “fall apart” if even a single important component dies or malfunctions.

Let us now try to briefly compare and contrast neural networks with the loosely coupled distributed systems. Indeed, distributed systems scale much better than their tightly coupled multiprocessor counterparts. Likewise, energy dissip-

²¹Of course, brains need some cooling, as well - in case of humans and advanced primates, chiefly through the mechanisms of respiration and perspiration.

tion is typically not a major concern in case of a distributed system, and, similarly, distributed systems degrade much more gracefully than tightly coupled multiprocessors. However, there are many other problems arising in and characteristic of the existing loosely coupled systems, as outlined in §1.3. In a highly effective neural network such as a human brain, some communication links may (and often do) fail, but due to an abundance of these links and multiple paths from one node to another, the system is rather robust to this type of failures. Moreover, all the issues and potential sources of trouble originating from the facts that different “nodes” in a distributed system can be (i) geographically very distant from one another, (ii) not very “compatible” with one another, and (iii) under control of different users with different, possibly conflicting and/or otherwise incompatible, individual computational goals, do not arise in case of human brains and other “real” neural networks at all. These problems, in the context of hypothetical biology- or physics-inspired (**ANN**-like or **CA**-like) parallel computer architectures of the future, would either not arise at all, or else be reasonably expected to be considerably simpler to solve than in the case of classical, coarse-grained distributed systems. Last but not least, while distributed systems scale better than the multiprocessor systems when it comes to adding more processing nodes, their scalability would still likely be many orders of magnitude behind that of the hypothetical parallel computers based on **ANN** or **CA** models. This scalability of the connectionist parallel models is largely due to the property of the fine-grain connectionist models that, as the number of processors grows, the delays related to fetching data from the “storage” need not grow proportionally to it - as there is no physically separated memory storage that each processing unit, however remote, would need to keep accessing during a computation run. Indeed, the increasing discrepancy between the processing and the memory access speeds in any von Neumannian architecture is simply not an issue in the **ANN**-like or **CA**-like models, and this feature alone hints at a considerable intrinsic advantage of the connectionist models when it comes to scalability.

To summarize, there are many potential important comparative advantages of the massively parallel computers based on the connectionist paradigms over the usual parallel or distributed extensions of the von Neumann digital computer models. While, in terms of the speed of performing various computational tasks, each of the two different parallel paradigms has its own “pool” of application domains for which it is much better suited than the competition, we have argued that, in terms of many other critical parameters, such as scalability as the number of processing units grows, robustness, graceful degradation, and energy dissipation, hypothetical fine-grain parallel computers of the future can be expected to convincingly outperform their classical, coarse-grain alternatives. Therefore, we are confident that the actual design (as opposed to mere simulation) of massively parallel computers inspired by some variants of **ANNs** or **CA** is a matter of *when* rather than *if*. The fine-grain parallel models and the actual man-designed computing systems based on the concepts and paradigms of these models do have the future - although, from the very early 21st century perspective, this future may not seem so near.

4. SUMMARY AND CONCLUSIONS

We have taken a computational view of the *connectionist models* that are mostly studied in other contexts, such as neural learning and artificial intelligence (**ANNs**), or dynamics of complex systems (**CA**), and then compared and contrasted these connectionist parallel computing models with the von-Neumann model based parallel architectures of the “classical” digital computers. In particular, as the connectionist models are capable of *massively parallel information processing*, they are therefore legitimate candidates for an alternative approach to design of highly parallel computers of the future - and, moreover, an alternative conceptually, architecturally and functionally rather remote from the traditional parallel and distributed computing systems.

We have argued that the connectionist models can be viewed as models of *very fine-grained parallelism*, as the processing units, and their basic operations, are much simpler than in case of the more familiar, *coarse(r)-grained* parallel computation models. Not only would conceivable computers designed according to the connectionist principles be expected to have several orders of magnitude more processors than any of the massively parallel coarse grain architectures that have ever been built, but such hypothetical fine-grained parallel computers would be very fundamentally different from the parallel digital computers as we know them today in almost every other respect. We have discussed some of the most profound differences between the fine-grain and the coarse-grain parallel models at both the architectural level (cf. the nature of processors and memory), and the functional level (how is the information represented, stored and processed). We have also argued that, in many respects, the actual (that is, physically realized) computers based on the connectionist paradigms and on the information-processing principles that we find in connectionist-like complex systems in Nature, could potentially exhibit several major advantages over the classical parallel and distributed computing systems.

While a truly fine-grain, massively parallel connectionist man-made computer is yet to be built, it can readily be argued that some concepts, ideas and paradigms stemming from the theoretical connectionist models studied herewith have already found some application in the recent and contemporary design of computer systems. For conciseness, we only mention a couple of examples from the area of hardware design. One well-known example is the *VLSI circuit design*. It appears very plausible that the connectionist concept of densely packing a very large number of elementary processing units with a great many communication connections between these units has already found its application in case of the *Very Large and Ultra Large System Integration designs (VLSI and ULSI, resp.)*. The relationship between VLSI (ULSI) and connectionist neural models has been studied since the 1980s (e.g., [AKFG], [SUBI]).

Another, more recent example of the potential utility of the connectionist paradigm is the *processor - in - memory* architecture design. The main motivation for embedding some fast, special purpose CPUs inside the memory is to partially overcome the growing gap between the CPU and the memory access speeds in contemporary electronic computers. It can be readily argued that, from an architectural viewpoint, this design technique represents a considerable departure from the usual von Neumann concept that memory and processors are clearly separated, both functionally and physically. However, the processors in(side of) mem-

ory design only partially overcomes the physical separation between processing and storage; functionally, the distinction between processors and memory remains. Therefore, neither of the two examples is a step toward truly implementing any of the connectionist models in the hardware but, rather, an illustration of how some concepts from the connectionist models have found application in the design of the “new generation” of non-connectionist digital computers or their components.

The state of the art when it comes to the connectionist models and their application to massively parallel computing is still in its infancy. While much of theoretical and experimental work has been done about properties of neural nets and cellular automata, the fact remains that, when studying the powers of these models, instead of implementing and, subsequently, evaluating the actual hardware that would be designed on the connectionist principles, we still actually only simulate workings of an ANN or a CA in software (that runs on some familiar, non-connectionist electronic digital computer). This, however, is in no way an indication that, viewed as conceptually realizable parallel computers, the connectionist models are either entirely unrealistic or obsolete; it only means that, at the current stage of science and technology, designing, implementing and, eventually, massively producing genuinely connectionist fine-grain parallel computers is still a too far fetched endeavor. However, promises that such parallel computers would hold, from scalability suitable for truly massive parallelism to robustness in the presence of noise to graceful degradation in the presence to failures to energy efficiency, will *eventually* make this endeavor more than worth while. For these reasons, we feel that connectionist fine-grain parallel models have a very promising future, not only in relatively specific areas such as machine learning or AI, but also as the underlying abstract models of the general-purpose massively parallel computers of tomorrow.

5. ACKNOWLEDGMENTS

The author is greatly indebted to Gul Agha, Tom Anastasio, Alfred Hubler and Sylvian Ray, all at University of Illinois. This work was partially supported by DARPA IPTO TASK Program, contract number F30602-00-2-0586, and DARPA MURI Program, contract number N00014-02-1-0715.

Bibliography

- [ADAM] A. Adamatzky, “Computing in Nonlinear Media and Automata”, IoP Publ., Bristol (UK), 2001
- [AKFG] L. A. Akers, D. K. Ferry, R. O. Grondin, “VLSI Implementations of Neural Systems”, in “The Computers and The Brain: Perspectives on Human and Artificial Intelligence”, J. R. Brink, C. R. Haden (eds.), Elsevier Sci. Publ. North Holland, 1989
- [ALKE] R. Allen, K. Kennedy, “Optimizing Compilers for Modern Architectures”, Morgan Kaufmann, 2001
- [ANAS] T. Anastasio, Introduction to neuroscience and computational brain theory, a lecture given in ANNCBT seminar, Beckman Institute, UIUC, Fall 2002
- [ASHB] R. B. Ashby, “Design for a Brain”, Wiley, 1960
- [BALL] D. H. Ballard, “An introduction to natural computation”, MIT Press, 1997
- [COKD] G. Coulouris, J. Dollimore, T. Kindberg, “Distributed Systems: Concepts and Design”, 3rd ed., Addison - Wesley, 2001
- [CHSE] P. S. Churchland, T. J. Sejnowski, ”The Computational Brain”, MIT Press, 1992
- [CORY] D. Cory, A. Fahmy, T. Havel, “NMR Spectroscopy: An Experimentally Accessible Paradigm for Quantum Computing”, 4th Workshop on Physics & Computation, Boston, 1996
- [DEMA] M. Delorme, J. Mazoyer (eds.), “Cellular Automata: A Parallel Model” Kluwer Academic Publ., 1999
- [GAJO] M. R. Garey, D. S. Johnson, “Computers and Intractability: A Guide to the Theory of NP-Completeness”, W. H. Freeman & Co., San Francisco, CA, 1979
- [GARZ] Max Garzon, “Models of Massive Parallelism: Analysis of Cellular Automata and Neural Networks”, Springer-Verlag, 1995
- [GISP] A. Gibbons, P. Spirakis, “Lectures on parallel computation”, Cambridge Int’l Series on Parallel Computation, Cambridge Univ. Press, 1993
- [GOMA] E. Goles, S. Martínez, “Neural and Automata Networks”, Kluwer Academic Publ., Dordrecht, 1990
- [GIRY] A. Gibbons, W. Rytter, “Efficient Parallel Algorithms”, Cambridge Univ. Press, 1988
- [HAYK] S. Haykin, “Neural networks : a comprehensive foundation”, Maxwell Macmillan Int’l, 1994
- [HOFF] A. Hoffmann, “Paradigms of Artificial Intelligence: A Methodological & Computational Analysis”, Springer - Verlag, Singapore, 1998
- [HOMU] J. E. Hopcroft, R. Motwani, J. D. Ullman, “Introduction to Automata Theory, Languages, and Computation”, 2nd ed., Addison-Wesley 2001
- [JACK] R. Jackendoff, “Consciousness and The Computational Mind”, Explorations in Cognitive Science series, MIT Press, 1990
- [KARA] R. M. Karp, V. Ramachandran, “Parallel Algorithms for Shared-Memory Machines”, Ch. §17 of “Handbook of Theoretical Computer Science”, Vol. I (ed. J. van Leeuwen), Elsevier Sci. Pub. & MIT Press, 1991
- [KOSE] C. Koch, I. Segev, “Methods in neuronal modeling: from synapses to networks”, MIT Press, 1989
- [MCPI] W. S. McCullough, W. Pitts, “A logical calculus for the ideas immanent in nervous activity”, Bull.Math. Biophysics, #5, 1942
- [MEPA] S. Melvin, Y. Patt, “Exploiting fine-grained parallelism through a combination of hardware and software techniques”, Proc. 18th Ann. Int’l. Symp. Comp. Arch., vol. 19, 1991
- [MULL] S. Mullender (ed.), “Distributed Systems”, 2nd ed., ACM Press & Addison-Wesley, 1993
- [NEU1] J. von Neumann, “The computers and the brain”, Yale Univ. Press, New Haven, 1958
- [NEU2] J. von Neumann, “Theory of Self-Reproducing Automata”, Univ. of Illinois Press, 1966
- [PAPA] C. Papadimitriou, “Computational Complexity”, Addison - Wesley, 1994
- [SEJN] T. J. Sejnowski, “ ‘The Computer and The Brain’ Revisited”, in “The Computers and The Brain: Perspectives on Human and Artificial Intelligence”, J. R. Brink, C. R. Haden (eds.), Elsevier Sci. Publ., North Holland, 1989
- [SIPS] M. Sipser, “Introduction to the Theory of Computation”, PWS Publishing Co., 1997
- [SRAY] S. Ray, lectures in CS442, “Artificial Neural Networks”, Dept. of Computer Science, UIUC, Fall 2001
- [SUBI] R. Suaya, G. Birtwistle (eds.), “VLSI and Parallel Computation”, Frontiers, Morgan Kaufmann Publ., 1990
- [TURI] A. M. Turing, “Computing machinery and intelligence”, in “Mind: A Quarterly Review of Psychology and Philosophy”, vol. LIX, No. 236, Oct. 1950
- [ULLM] J. D. Ullman, “Computational Aspects of VLSI”, Comp. Sci. Press, 1984
- [WOLF] S. Wolfram, “Cellular Automata and Complexity: Collected Papers”, Addison - Wesley, 1994
- [XAIY] C. Xavier, S. S. Iyengar, “Introduction to Parallel Algorithms”, Wiley Series on Parallel & Distributed Computing, (ed. A. Y. Zomaya), J. Wiley & Sons, 1998
- [ZEOO] A. Zomaya, F. Ercal, S. Olariou (eds.), “Solutions to Parallel and Distributed Computing Problems: Lessons from Biological Sciences”, Wiley Series on Parallel & Distributed Computing, (ed. A. Y. Zomaya), J. Wiley & Sons, 2001

Concurrency vs. Sequential Interleavings in 1-D Threshold Cellular Automata

Predrag Tomic*, Gul Agha

Open Systems Laboratory, Department of Computer Science

University of Illinois at Urbana-Champaign, 201 N. Goodwin, Urbana, IL 61801 (USA)

{p-tomic, agha}@cs.uiuc.edu

Abstract

Cellular automata (**CA**) are an abstract model of fine-grain parallelism, as the node update operations are rather simple, and therefore comparable to the basic operations of the computer hardware. In a classical **CA**, all the nodes execute their operations in parallel, that is, (logically) simultaneously. We consider herewith the sequential version of **CA**, or **SCA**, and compare it with the classical, parallel **CA**. In particular, we show that there are **1-D CA** with very simple node state update rules that cannot be simulated by any comparable **SCA**, irrespective of the node update ordering. While the result is trivial if one considers a single computation on a chosen input, we find it both nontrivial, and having some important and far-reaching implications when applied to all possible inputs and, moreover, to the entire nontrivial classes of **CA** (**SCA**). We also share some thoughts on how to extend our results herein, and we try motivate the study of genuinely asynchronous cellular automata.

1. Introduction and Motivation

Cellular automata (**CA**) were originally introduced as abstract mathematical models that can capture, at a high level, the behavior of biological systems capable of self-reproduction [15]. Subsequently, **CA** have been extensively studied in a great variety of application domains, but mostly in the context of physics and, more specifically, of studying complex (physical or biological) systems and their dynamics (e.g., [20-22]). However, **CA** can also be viewed as an abstraction of massively parallel computers (e.g., [7]). Herein, we study a particular simple yet nontrivial class of **CA** from a computer science perspective. This class are the *threshold cellular automata*. We pose (and partially answer) some fundamental questions regarding the nature of the **CA** parallelism, i.e., the concurrency of the classical **CA** computation; the analysis is done in the context of *threshold CA*.

Namely, it is well known that **CA** are an abstract computational model of *fine-grain parallelism* [7], in that the elementary operations executed at each node are rather simple and hence comparable to the basic operations performed by the computer hardware - yet, due to interaction and synergy among a (typically) great number of these nodes, many **CA** are capable of highly complex behaviors (i.e., computations). In a classical (parallel) **CA**, whether finite or infinite, all the nodes execute their operations *logically simultaneously*: in general, the state of node x_i at time step $t + 1$ is some simple function of the states of node x_i itself, and a set of its pre-specified neighbors at time t .

We consider herewith the sequential version of **CA**, or **SCA**, and compare it with the classical, *parallel (concurrent)* **CA**. In particular, we show that there are **1-D CA** with very simple node state update rules that cannot be simulated by any comparable **SCA**, irrespective of the node update ordering. While the result is trivial if one considers a single **CA**, we find the result quite nontrivial, important and with some far-reaching implications when applied to the entire classes of **CA** and **SCA**. Hence, granularity of the basic **CA** operations, insofar as the ability to simulate their concurrent computation via appropriate nondeterministic sequential interleavings of these basic operations, turns out not to be quite *fine enough*, as we prove that no such analogue of the sequential interleaving semantics applied to concurrent programs of classical **CA** can capture even rather simplistic parallel **CA** computations. We also share some thoughts on how to extend the results presented herein, and, in particular, we try motivate the study of *genuinely asynchronous cellular automata*, where asynchrony applies not only to the local computations at individual nodes, but also to *communication* among different nodes (via “shared variables” stored as the respective nodes’ states).

An example of asynchrony in the local node updates (i.e., asynchronous computation at different “processors”) is the case when, for instance, the individual nodes update one at the time, according to some random order. This is a kind of asynchrony found in the literature, e.g., in [10]. It is important to understand, however, that even in case of what

is referred to as *asynchronous cellular automata* (*ACA*) in the literature, the term *asynchrony* there applies to local updates (i.e., computations) *only*, but not to communication, since a tacit assumption of the globally accessible global clock still holds. We prefer to refer to this kind of (weakly asynchronous) (*A*)*CA* as *sequential cellular automata*, and, in this work, consistently keep the term *asynchronous cellular automata* for those *CA* that do not have a global clock (see §4).

Before dwelling into the issue of parallelism vs. arbitrary sequential interleavings applied to threshold cellular automata, we first clarify the terminology, and then introduce the relevant concepts through a simple programming exercise in §§1.1.

An important remark is that we use the terms *parallel* and *concurrent* as synonyms throughout the paper. This is perhaps not the most standard convention among the researchers of programming languages semantics and semantic models of concurrency (e.g., [17], [16]), but we are not alone in not making the distinction between the two notions (cf. discussion in [16]). Moreover, by a *parallel* (equivalently, *concurrent*) *computation* we shall mean actions of several processing units that are carried out *logically* (if not necessarily *physically*) *simultaneously*. In particular, when referring to parallel (or, equivalently, concurrent) computation, we do assume a perfect synchrony. This approach is primarily motivated by the nature of *CA* “hardware” and the way classical *CA* compute.

1.1. Capturing concurrency by nondeterministic sequential interleavings

While our own brains are massively parallel computing devices, we seem to think and function rather sequentially. Indeed, human mind’s approach to problem solving is usually highly sequential. In particular, when designing an algorithm or writing a computer program that is inherently parallel, we prefer to be able to understand such an algorithm or program in the sequential terms. It is not surprising, therefore, that since the very beginning of the design of parallel algorithms and parallel computation formalisms, a great deal of research effort has been devoted to interpreting parallel computation in the more familiar, sequential terms. One of the most important contributions in that respect is the nondeterministic sequential *interleaving semantics* of concurrency [14].

When interpreting concurrency via interleaving semantics, a natural question arises: *Given a parallel computing model, can its concurrent execution always be captured by such sequential nondeterminism, so that any given parallel computation can be faithfully reproduced via an appropriate choice of a sequential interleaving of the operations involved?* The answer is “Yes”, For most theoreticians of

parallel computing (that is, all the “believers” in interleaving semantics as contrasted with, e.g., proponents of *true concurrency*, an alternative model not discussed herewith), the answer is apparently “Yes” - provided that we simulate concurrent execution via sequential interleavings at a sufficiently high level of granularity of the basic computational operations. However, it need not always be clear, how do we tell, given a parallel computation in the form of a set of concurrently executing instructions or processes, if the particular level of granularity is *fine enough*, i.e., whether the operations at that granularity level can truly be rendered *atomic* for the purpose of capturing concurrency via sequential interleavings?

We shall illustrate the concept of *sequential interleaving semantics* of concurrency with a simple example. Let’s consider the following trivia question from a sophomore parallel programming class: *Find a simple example of two instructions such that, when executed in parallel, they give a result not obtainable from any corresponding sequential execution sequence?*

A possible answer: Assume $x = 0$ initially and consider the following two programs

$x \leftarrow x + 1; x \leftarrow x + 1$

vs.

$x \leftarrow x + 1 \parallel x \leftarrow x + 1$

Sequentially, one *always* gets the same answer: $x = 2$. In parallel (that is, if the two assignment operations to the same variable x are done concurrently), however, one gets $x = 1$. It appears, therefore, that no sequential ordering of operations can reproduce parallel computation - at least not at the granularity level of high-level instructions as above.

The whole “mystery” is resolved if we look at the possible sequential executions of the corresponding machine instructions:

LOAD $x, *m$	LOAD $x, *m$
ADD $x, \#1$	ADD $x, \#1$
STORE $x, *m$	STORE $x, *m$

There certainly exists a choice of a *sequential interleaving* of the six machine instructions above that leads to “*parallel*” behavior (i.e., the one where, after the code is executed, $x = 1$); in fact, there are several such permutations of instructions. Thus, by refining granularity from a high-level language instructions down to machine instructions, we can certainly preserve the interleaving “semantics” of concurrency, as the high-level language “concurrent” computation can be perfectly well understood in terms of the sequential interleavings of computational operations at the level of assembly language instructions.

2. Cellular Automata and Types of Their Configurations

We introduce *classical CA* by first considering (*deterministic*) *Finite State Machines (FSMs)* such as *Deterministic Finite Automata (DFA)*. An *FSM* has finitely many states, and is capable of reading input signals coming from the outside. The machine is initially in some starting state; upon reading each input signal (a single binary symbol, in the standard *DFA* case), the machine changes its state according to a pre-defined and fixed rule. In particular, the entire memory of the system is contained in what “current state” the machine is in, and nothing else about previously processed inputs is remembered. Hence, the probabilistic generalization of deterministic *FSMs* leads to (discrete) Markov chains. It is important to notice that there is no way for a *FSM* to overwrite, or in any other way affect the input data stream. Thus *individual FSMs* are computational devices of rather limited power.

Now let us consider many such *FSMs*, all identical to one another, that are lined up together in some regular fashion, e.g., on a straight line or a regular 2-D grid, so that each single “node” in the grid is connected to its immediate neighbors. Let’s also eliminate any external sources of input streams to the individual machines at the nodes, and let the current values of any given node’s neighbors be that node’s only “input data”. If we then specify the set of the possible values held in each node (typically, this set is $\{0, 1\}$), and we also identify this set of values with the set of the node’s *internal states*, we arrive at an informal definition of a classical cellular automaton. To summarize, a *CA* is a finite or infinite regular grid in one-, two- or higher-dimensional space, where each node in the grid is a *FSM*, and where each such node’s input data at each time step are the corresponding internal states of the node’s neighbors. Moreover, in the most important special case - the Boolean case, this *FSM* is particularly simple, i.e., it has only two possible internal states, 0 and 1. All the nodes of a classical *CA* execute the *FSM* computation in unison, i.e., (*logically*) *simultaneously*. We note that infinite *CA* are capable of universal (Turing) computation, and, moreover, are actually strictly more powerful than classical Turing machines (e.g., [7]).

More formally, we follow [7] and define classical (that is, synchronous and concurrent) *CA* in two steps: by first defining the notion of a *cellular space*, and subsequently that of a *cellular automaton* defined over an appropriate cellular space.

Definition 1: *Cellular Space* Γ is an ordered pair (G, Q) where G is a regular graph (finite or infinite), and Q is a finite set of states that has at least two elements, one of which being the special *quiescent state*, denoted by 0.

Definition 2: *Cellular Automaton* A is an ordered triple (Γ, N, M) where:

- Γ is a *cellular space*;
- N is a *fundamental neighborhood*;
- M is a *finite state machine* such that the input alphabet of M is $Q^{|N|}$, and the local transition function (update rule) for each node is of the form $\delta : Q^{|N|+1} \rightarrow Q$ for *CA with memory*, and $\delta : Q^{|N|} \rightarrow Q$ for *memoryless CA*.

The local transition rule δ specifies how each node updates, based on its current value and that of its neighbors in N . By composing local transition rules for all nodes together, we obtain the *global map* on the set of (global) configurations of a cellular automaton.

Assuming a large number of nodes, there is plenty of potential for parallelism in the *CA* hardware. Actually, classical *CA* defined over infinite cellular spaces provide *unbounded parallelism* where, in particular, an infinite amount of information processing is carried out in a finite time (even in a single “parallel” step). Roughly, the underlying cellular space corresponds to the *CA* “hardware”, whereas the *CA* “software” or program is given by the local update rule δ . The global evolution (or, analogously, massively parallel computation) of a *CA* is then obtained by the composition of the effects of the local node update rule to each of the nodes.

We now change pace and introduce some terminology borrowed from physics that we find appropriate and useful for characterizing *all possible computations* of a parallel or sequential *CA*. To this end, a (*discrete*) *dynamical system* view of *CA* is helpful. A *phase space* of a dynamical system is a (finite or infinite) directed graph where the vertices are the *global configurations* (or *global states*) of the system, and directed edges correspond to possible transitions from one global state to another.

As for any other kind of dynamical systems, we can define the fundamental, qualitatively distinct types of (global) configurations that a cellular automaton can find itself in. The classification below is based on answering the following question: starting from a given global *CA* configuration, can the automaton return to that same configuration after a finite number of (parallel) computational steps?

Definition 3: A *fixed point (FP)* is a configuration in the phase space of a *CA* such that, once the *CA* reaches this configuration, it stays there forever. A *cycle configuration (CC)* is a state that, if once reached, will be revisited infinitely often with a fixed, finite period of 2 or greater. A *transient configuration (TC)* is a state that, once reached, is never going to be revisited again.

In particular, a *FP* is a special, degenerate case of *CC* with period 1. Due to deterministic evolution, any configuration of a classical, parallel *CA* is either a *FP*, a proper *CC*, or a *TC*.

3. 1-D CA vs. SCA Comparison and Contrast for Simple Threshold Rules

After the introduction, motivation and the necessary definitions, we now proceed with our main results and their meaning. Technical results, and their proofs, are given in this section; discussion of the implications and relevance of these results, as well as the possible generalizations and extensions, will follow in *Section 84*.

Herein, we compare and contrast the classical, concurrent *CA* with their sequential counterparts, *SCA*, in the context of the simplest (nonlinear) local update rules possible, viz., the *CA* in which the nodes locally update according to *linear threshold functions*. Moreover, we choose these threshold functions to be *symmetric*, so that the resulting *CA* are also *totalistic* (see, e.g., [7] or [22]). We show the fundamental difference in the configuration spaces, and therefore possible computations, in case of the classical, concurrent automata on one, and the sequential threshold cellular automata, on the other hand: while the former can have temporal cycles (of length two), the computations of the latter, under some mild additional conditions whose sole purpose is to ensure *some form of convergence*, *always* converge to a fixed point.

We fix the following conventions and terminology. Throughout, only *Boolean CA* and *SCA* are considered; in particular, the set of possible states of any node is $\{0, 1\}$. The terms “monotone symmetric” and “symmetric (linear) threshold” functions/update rules/automata are used interchangeably; similarly, “(global) dynamics” and “computation”, when applied to any kind of an automaton, are used synonymously. Unless stated otherwise, *CA* denotes a classical, concurrent cellular automaton, whereas a cellular automaton where the nodes update sequentially is always denoted by *SCA*. Also, unless explicitly stated otherwise, *CA (SCA) with memory* are assumed, and the default cellular space is a two-way infinite line. Moreover, all the underlying cellular spaces throughout the next two subsections are (finite or infinite) lines or rings.¹ The terms “phase space” and “configuration space” are used synonymously, as well, and sometimes abridged to *PS* for brevity.

3.1. A simple motivating example

A *1-D cellular automaton of radius r* is a *CA* defined over a one-dimensional string of nodes, such that each node’s next state depends on the current states of its neighbors to the left and right that are no more than r nodes away (and, in case of *CA with memory*, on the current state of that node itself). In case of a *Boolean CA with memory*,

¹We have already generalized some of the results that follow to more general cellular spaces, but, for the reasons of conciseness and clarity of exposition, those results will not be discussed herein.

therefore, each node’s next state depends on $2r + 1$ input bits, while in the *memoryless case*, the local update rule is a function of $2r$ input bits. The string of nodes can be a finite line graph, a ring (corresponding to “circular boundary conditions”), a one-way infinite string, or, in the most common case one finds in the literature, the cellular space is a two-way infinite string.

We compare and contrast the qualitative properties of configurations spaces, and therefore dynamics or possible computations, of the classical, parallel *CA* versus the dynamics (computations) of *SCA*. Sequential cellular automata (*SCA*) and their generalizations to non-regular (finite) graphs have been already studied in the context of a formal theory of computer simulation (see, e.g., [2-6]).

There are plenty of simple, even trivial examples where not only are concrete computations of parallel *CA* from particular initial configurations different from the corresponding computations of the sequential *CA*, but actually the entire configuration spaces of the *parallel CA* and the corresponding *SCA* are structurally rather different.

As one of the simplest examples conceivable, consider a trivial *CA* with more than one node (so that talking about “parallel computation” makes sense), namely, a two-node *CA* where each node computes the Boolean *XOR* of the two inputs (viz., of the node’s own current state, and that of its only neighbor). The two phase spaces are given in *Fig. 1*. In (b), since the corresponding automaton is *nondeterministic*, the numbers next to the transition arrows indicate which node, 1 or 2, is updating and thus causing the indicated global state transition from the current state.

In the parallel case, the state 00 is the “sink”, and the entire configuration space is as in *Fig. 1 (a)*. So, regardless of the starting configuration, after at most two parallel steps, the fixed point “sink” state, that is, in the language of *nonlinear dynamics*, a stable global attractor, will be reached.

In case of the sequential node updates, the configuration 00 is still a FP but, this time, it is not reachable from any other configuration. Moreover, there are two more (unstable) pseudo-fixed points, 01 and 10, and two temporal two-cycles. In particular, while all three states, 11, 10 and 01, are *transient states* in the parallel case, sequentially, each of them, for any “typical” sequence of node updates, is going to be revisited, either as a pseudo-FP or a cycle state. In fact, for some sequences of node updates such as, e.g., $(1, 1, 2, 2, 1, 2, 2, 1, \dots)$, configurations 01 and 10 are *both* pseudo-fixed point states and cycle states. The phase space capturing all possible sequential computation is given in *Fig. 1 (b)*.

Several observations are in order. First, overall, the sequential configuration space seems richer than its parallel counterpart. In particular, whereas, due to determinism, any FP state of a classical *CA* is necessarily a stable attractor or

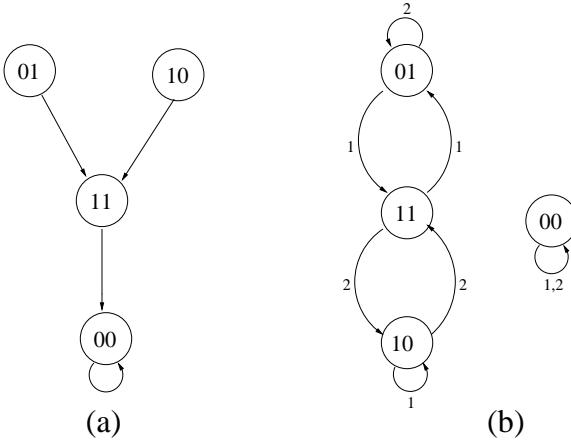


Figure 1. Configuration spaces for two-node (a) parallel and (b) sequential XOR CA, resp.

a “sink” (in the terminology of *complex dynamics*), and, in particular, once a FP configuration is reached, all the future iterations stay there, in case of different possible sequential computations on the same cellular space, pseudo-fixed points clearly need not be stable. Also, whereas the phase space of a parallel CA is cycle-free (if we do not count FPs as “proper” cycles), the phase space of the corresponding SCA has nontrivial temporal cycles. On the other hand, the union of all possible sequential computations (“interleavings”) cannot fully capture the concurrent computation, either: e.g., consider *reachability* of the state 00.

These properties can be largely attributed to a relative complexity of the *XOR* function as the update rule, and, in particular, to *XOR*’s *non-monotonicity*. They can also be attributed to the idiosyncrasy of the example chosen. In particular, temporal cycles in the sequential case are not surprising. Also, if one considers CA on say four nodes with circular boundary conditions (that is, a CA ring on four nodes), these XOR CA do have nontrivial cycles in the parallel case, as well. Hence, for XOR CA with sufficiently many nodes, the types of computations that parallel CA and sequential SCA are capable of, are quite comparable. Moreover, in cases when one is richer than the other, it seems reasonable that SCA will be capable of more interesting computations than parallel CA, given the nondeterminism due to all the possibilities for node update sequences.

This detailed discussion of a trivial example of the (S)CA phase spaces has the main purpose of motivating what is to follow: an entire class of CA and SCA, with the node update functions even simpler than *XOR*, yet for which, irrespective of the number of nodes, the boundary conditions and other details, it is the concurrent CA that are *provably* capable of computations that no corresponding (or similar, in a sense to be defined below) SCA are capable of.

3.2. On the existence of cycles in threshold CA and SCA

Now we consider the threshold automata in parallel and sequential settings.

Definition 4: A *threshold automaton (threshold CA)* is a (parallel or sequential) cellular automaton where δ is a (herein, Boolean-valued) *linear threshold function*.

Herein, we also assume δ to be a *symmetric function* of all of its inputs.

Due to the nature of the node update rules, cyclic behavior intuitively should not be expected in such automata. This is, generally, (almost) the case, as will be shown below. We argue that the importance of the results in this subsection largely stems from the following three factors: (i) the local update rules are the simplest nonlinear totalistic rules one can think of; (ii) given the rules, the cycles are not to be expected - yet they exist, and in case of classical, parallel CA *only*; and, related to that observation, (iii) it is, for this class of (S)CA, the parallel CA that exhibit the more interesting behavior than sequential SCA, and, in particular, while there is nothing (qualitatively) among the possible sequential computations that is not present in the parallel case, the classical parallel threshold CA are capable of “oscillatory/non-converging computations” - they may have nontrivial temporal cycles - that cannot be reproduced by any threshold SCA.

The results below hold for two-way infinite 1-D CA, as well as for finite CA and SCA with sufficiently many nodes and circular boundary conditions (i.e., for (S)CA whose cellular spaces are finite rings).

Lemma 1:

(i) A 1-D classical (i.e., parallel) CA with $r = 1$ and the MAJORITY update rule has (finite) temporal cycles in the phase space (PS).

(ii) 1-D Sequential CA with $r = 1$ and the MAJORITY update rule do not have any (finite) cycles in the phase space, *irrespective* of the sequential node update order ρ .

Remark: In case of infinite sequential SCA as in the Lemma above, a nontrivial cycle configuration does not exist even in the limit. We also note that ρ is an arbitrary sequence of an SCA nodes’ indices, not necessarily a (finite or infinite) permutation.

Proof: To show (i), we exhibit an actual two-cycle. Consider either an infinite 1-D CA, or a finite one, with circular boundary conditions and an even number of nodes, $2n$. Then the configurations $(10)^\omega$ and $(01)^\omega$ in the infinite case ($(10)^n$ and $(01)^n$ in the finite ring case) form a 2-cycle.

To prove (ii), we must show that no cycle is ever possible, irrespective of the starting configuration. We consider all possible 1-neighborhoods (there are eight of them: 000, 001, ..., 111), and show that, locally, none of them

can be cyclic yet not fixed. The case analysis is simple: 000 and 111 are stable (fixed) sub-configurations. Configuration 010, after a single node update, can either stay fixed, or else evolve into any of $\{000, 110, 011\}$; since we are only interested in non-FPs, in the latter case, one can readily show by induction that, after any number of steps, the only additional sub-configuration that can be reached is 111, i.e., assuming 010 is not fixed, $010 \rightarrow^* \{000, 110, 011, 111\}$. However, $010 \notin \{000, 110, 011, 111\}$. By symmetry, similar analysis holds for sub-configuration 101. On the other hand, 110 and 011 either remain fixed, or else at some time step t evolve to 111, which is a fixed point. Similar analysis applies to 001 and 100. Hence, no local neighborhood $x_1x_2x_3$, once changed, can ever “come back”. Therefore, there are no proper cycles in Sequential 1-D CA with $r = 1$ and $\delta = \text{MAJORITY}$.

Part (ii) of the Lemma above can be readily generalized: even if we consider local update rules δ other than the *MAJORITY* rule, yet restrict δ to *monotone symmetric Boolean functions* of the input bits, such sequential CA still do not have any proper cycles.

Theorem 1: For any *Monotone Symmetric Boolean 1-D Sequential CA* A with $r = 1$, and any sequential update order ρ , the phase space $PS(A)$ of the automaton A is cycle-free.

Similar results to those in Lemma 1 and Theorem 1 also hold for *1-D CA* with radius $r = 2$:

Lemma 2:

- (i) *1-D (parallel) CA* with $r = 2$ and with the *MAJORITY* node update rule have (finite) cycles in the phase space.
- (ii) Any *1-D SCA* with *MAJORITY* node update rule, $r = 2$ and any sequential order on node updates has a cycle-free phase space.

Generalizing Lemmata 1 and 2, part (i), we have the following

Corollary 1: For all $r \geq 1$, there exists a *monotone symmetric CA* (that is, a *threshold automaton*) A such that A has (finite) cycles in the phase space.

Namely, given any $r \geq 1$, a (classical, concurrent) CA with $\delta = \text{MAJORITY}$ has at least one two-cycle in the PS : $\{(0^r 1^r)^\omega, (1^r 0^r)^\omega\}$. If $r \geq 3$ is odd, then such a threshold automaton has at least two distinct two-cycles, since $\{(01)^\omega, (10)^\omega\}$ is also a two-cycle. Analogous results hold for *threshold CA (SCA)* defined on finite 1-D cellular spaces, provided that such automata have sufficiently many nodes and assuming circular boundary conditions (i.e., assuming Γ is a sufficiently big finite ring). Moreover, the result extends to finite and infinite CA in higher dimensions, as well; in particular, *2D rectangular grid CA* and *Hypercube CA* with $\delta = \text{MAJORITY}$ (or another nontrivial symmetric threshold update rule) have two-cycles in their respective phase spaces.

More generally, for any underlying cellular space Γ that is a (finite or infinite) *bipartite graph*, the corresponding (nontrivial) *threshold parallel CA* have temporal two-cycles.

It turns out that the two-cycles in the PS of concurrent CA with $\delta = \text{MAJORITY}$ are actually the only type of (proper) temporal cycles such cellular automata can have. Indeed, for any *symmetric linear threshold update rule* δ , and any (finite or infinite) regular Cayley graph as the underlying cellular space, the following general result holds [7], [8]:

Proposition 1: Let a classical CA $A = (\Gamma, N, T)$ be such that T is an elementary symmetric threshold local update rule applied to a finite cellular space Γ . Then for all configurations $C \in PS(A)$, there exists $t \geq 0$ such that $T^{t+2}(C) = T^t(C)$.

In particular, this result implies that, in case of any (finite) monotone symmetric automaton, for any starting configuration C_0 , there are only two possible kinds of orbits: upon repeated iteration, after finitely many steps, the computation either converges to a fixed point configuration, or else one arrives at a two-cycle.

It is almost immediate that, if we allow the underlying cellular space Γ to be infinite, if computation from a given starting configuration converges at all, it will converge either to a fixed point or a two-cycle (but never to a cycle of, say, period three - or any other finite period). The result also extends to finite and infinite SCA, provided that we reasonably define what is meant by a single computational step in a situation where the nodes update one at a time.²

To summarize, symmetric linear threshold 1-D CA, depending on the starting configuration, may converge to a fixed point or a temporal two-cycle; in particular, they may end up “looping” in finite (but nontrivial) temporal cycles. In contrast, the corresponding classes of SCA can never cycle; while for simplicity we have shown that this holds only for SCA with short-range interactions (small r), the result actually holds for (finite) SCA with arbitrary finite radii of interaction, $r \geq 1$. In particular, given any sequence of node updates of a finite threshold SCA, if this sequence satisfies an appropriate *fairness condition* then it can be shown that the computation of such a threshold SCA is guaranteed to converge to a fixed point. Since this holds irrespective of the choice of the sequential update ordering (and, extending to infinite SCA, temporal cycles cannot be obtained even “in the limit”, that is, via infinitely long computations, obtained

² Additionally, in order to ensure some sort of convergence of a given SCA, and, more generally, in order to ensure that, in some sense, *all* nodes get a chance to update their states, an appropriate condition that guarantees *fairness* may need to be specified. For finite SCA, one sufficient such condition is to impose a fixed upper bound on the number of sequential steps before any given node gets its “turn” to update. In infinite SCA case, the issue of fairness is nontrivial, and some form of *dove-tailing* of sequential individual node updates may need to be imposed; further discussion of this issue, however, is beyond our current scope.

by allowing arbitrary infinite sequences of individual node updates), we conclude that no choice of “sequential interleaving” can capture the concurrent computation. Consequently, the “interleaving semantics” of *SCA* fails to fully capture the concurrent behavior of classical *CA* even for this, simplest nonlinear class of totalistic *CA*, namely, the symmetric threshold cellular automata.

4 Discussion and Future Directions

The results in §3 show that, even for the very simplest (nonlinear) totalistic cellular automata, nondeterministic interleavings dramatically fail to capture concurrency. It is not surprising that one can find a (classical, concurrent) *CA* such that no sequential *CA* with the same underlying cellular space and the same node update rule can reproduce identical or even “isomorphic” computation, as the example at the beginning of §3 clearly shows (see *Fig. 1* and the related discussion). However, we find it rather interesting that very profound differences (a possibility of looping vs. the guaranteed convergence to a fixed point configuration) can be observed in case of the simplest nonlinear 1-D parallel and sequential *CA* with symmetric threshold functions as the node update rules, and that this profound difference does not apply merely to individual (*S*)*CA* and/or their particular computations, but the possible computations of the entire class of the (*symmetric*) *threshold CA* update rules.

Moreover, the differences in parallel and sequential computations in case of the Boolean *XOR* local update rule can be largely ascribed to the properties of the *XOR* function. For instance, given that *XOR* is not *monotone*, the existence of temporal cycles is not at all surprising. In contrast, monotone functions such as *MAJORITY* are intuitively expected not to have cycles, i.e., in case of finite domains and converging computations, to always converge to a fixed point. This intuition about the monotone symmetric *SCA* is shown correct. It is actually, in a sense, (statistically) “almost correct” in case of the parallel *CA*, as well, in that the actual non-FP cycles can be shown to be very few, and without any incoming transients [19]. Thus, in this case, the very existence of the (rare) nontrivial temporal cycles can be ascribed directly to the assumption of *perfect synchrony* (i.e., effective simultaneity) of the parallel node updates.

We now briefly discuss some possible extensions of the results presented thus far. In particular, we are considering extending our study to *non-homogeneous threshold CA*, where not all the nodes necessarily update according to *one and the same* threshold update rule. Another obvious extension is to consider 2-D and other higher-dimensional threshold *CA*, as well as *CA* defined over regular Cayley graphs that are not simple Cartesian grids. It is also of interest to consider *CA*-like finite automata defined over *arbitrary* rather than only regular (finite) graphs. We already have

some results along these two lines, but do not include them herein due to space constraints.

Another interesting extension is to consider classes of node update rules beyond the threshold functions. One obvious candidate are the monotone functions that are not necessarily symmetric (that is, such that the corresponding *CA* need not be totalistic or semi-totalistic). A possible additional twist, as mentioned above, is to allow for different nodes to update according to different monotone (symmetric or otherwise) local update rules. At what point of the increasing automata complexity, if any, the possible sequential computations “catch up” with the concurrent ones appears an interesting problem to consider.

Yet another direction for further investigation is to consider other models of (a)synchrony in cellular automata. We argue that the classical concurrent *CA* can be viewed, if one is interested in node-to-node interactions among the nodes that are not close to one another, as a class of computational models of *bounded asynchrony*. Namely, if nodes x and y are at distance k (i.e., k nodes apart from each other), and the radius of the *CA* is r , then any change in the state of y can affect the state of x no sooner, but also and more importantly, *no later* than after about $\frac{k}{r}$ (parallel node update) computational steps. As the nodes all update “in sink”, locally a classical *CA* is a perfectly synchronous concurrent system. However, globally, i.e., if one is interested in the interactions of nodes that are at a distance greater than r apart, then the classical *CA* and their various graph automata extensions are a class of models of *bounded asynchrony*.

We remark that two particular classes of graph automata defined over arbitrary (not necessarily regular, or Cayley) *finite* graphs, viz., sequential and synchronous dynamical systems (SDSs and SyDSs, respectively), and their various phase space properties, have been extensively studied; see, e.g., [3-6] and references therein. It would be interesting, therefore, to consider *asynchronous cellular and graph automata*, where the nodes are not assumed any longer to update in unison and, moreover, where no global clock is assumed. We again emphasize that such automata would entail what can be viewed as *communication asynchrony*, thus going beyond the kind of asynchrony in computation at different nodes (that is, beyond the arbitrary sequential node updates yet with respect to the global time) that has been studied since at least 1984 (e.g., [10], [11]).

We propose a broad study of the general phase space properties, and a qualitative comparison-and-contrast of the asynchronous *CA* (*ACA*) and the classical *CA* and *SCA*. Such a study could shed light on detecting computational behaviors that are solely due to asynchrony, that is, what can be viewed as an abstracted version of “network delays” in physically realistic (asynchronous) cellular automata. Communication asynchronous *CA*, i.e., various nondeterministic choices for a given (*A*)*CA* that are due to asyn-

chrony, can be shown to subsume all possible behaviors of classical and sequential *CA* with the same corresponding (Γ, N, M) . In particular, the nondeterminism that arises from (unbounded) asynchrony subsumes the nondeterminism of a kind studied in §3; but the question arises, exactly how much more expressive and powerful the former model really is than the latter.

5 Conclusions

We present herein some early steps in studying cellular automata when the unrealistic assumptions of *perfect synchrony* and *instantaneous unbounded parallelism* are dropped. Motivated by the well-known notion of the sequential interleaving semantics of concurrency, we try to apply this concept to parallel *CA* and thus motivate the study of sequential cellular automata, *SCA*, and the comparison and contrast between *SCA* and classical, concurrent *CA*. Concretely, we show that, even in very simplistic cases, this sequential semantics fails to capture concurrency of classical *CA*. Hence, simple as they may be, the basic operations (local node updates) in classical *CA* cannot always be considered atomic. In particular, the fine-grain parallelism of *CA* turns out not to be quite fine enough when it comes to capturing concurrent execution via nondeterministic sequential interleavings of those basic operations. It then seems reasonable to consider a single local node update to be made of an ordered sequence of finer elementary operations: (i) fetching (“receiving”?) all the neighbors’ values, (ii) updating one’s own state according to the update rule δ , and (iii) making available (“sending”?) one’s new state to the neighbors.

Motivated by these early results on sequential and parallel *CA* and their implications, we next consider various extensions. The main idea is to introduce a class of *genuinely asynchronous CA*, and formally study their properties. This would hopefully yield, down the road, some significant insights into the fundamental issues related to bounded vs. unbounded asynchrony, formal sequential semantics for parallel and distributed computation, and, on the *CA* side, to identification of those classical parallel *CA* phase space properties that are a direct consequence of the (physically unrealistic) assumption of perfectly synchronous and simultaneous node updates.

We also argue that various extensions of the basic *CA* model can provide a simple, elegant and useful framework for a high-level study of various global qualitative properties of distributed, parallel and real-time systems at an abstract and rigorous, yet comprehensive level.

Acknowledgments: The work presented herein was supported by the DARPA IPTO TASK Program, contract number F30602-00-2-0586. Many thanks to Reza Ziae (UIUC) for several useful discussions.

Bibliography

- [1] R. B. Ashby, ‘Design for a Brain’, Wiley, 1960
- [2] C. Barrett and C. Reidys, ‘Elements of a theory of computer simulation I: sequential CA over random graphs’, Applied Math. & Comput., vol. 98 (2-3), 1999
- [3] C. Barrett, H. Hunt, M. Marathe, S. S. Ravi, D. Rosenkrantz, R. Stearns, and P. Tomic, ‘Gardens of Eden and Fixed Points in Sequential Dynamical Systems’, Discrete Math. & Theoretical Comp. Sci. Proc. AA (DM-CCG), July 2001
- [4] C. Barrett, H. B. Hunt III, M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, R. E. Stearns, ‘Reachability problems for sequential dynamical systems with threshold functions’, TCS 1-3: 41-64, 2003
- [5] C. Barrett, H. Mortveit, and C. Reidys, ‘Elements of a theory of computer simulation II: sequential dynamical systems’, Applied Math. & Comput. vol. 107(2-3), 2000
- [6] C. Barrett, H. Mortveit, and C. Reidys, ‘Elements of a theory of computer simulation III: equivalence of sequential dynamical systems’, Appl. Math. & Comput. vol. 122(3), 2001
- [7] M. Garzon, ‘Models of Massive Parallelism: Analysis of Cellular Automata and Neural Networks’, Springer, 1995
- [8] E. Goles, S. Martinez, ‘Neural networks: theory and applications’, Kluwer, Amsterdam, 1990
- [9] E. Goles, S. Martinez (eds.), ‘Cellular Automata and Complex Systems’, Nonlinear Phenomena and Complex Systems series, Kluwer, Dordrecht, 1999
- [10] T. E. Ingerson and R. L. Buvel, ‘Structure in asynchronous cellular automata’, Physica D: Nonlinear Phenomena, vol. 10 (1-2), Jan. 1984
- [11] S. A. Kauffman, ‘Emergent properties in random complex automata’, ibid.
- [12] R. Milner, ‘A Calculus of Communicating Systems’, Lecture Notes Comp. Sci., Springer, Berlin, 1989
- [13] R. Milner, ‘Calculi for synchrony and asynchrony’, Theoretical Comp. Sci. 25, Elsevier, 1983
- [14] R. Milner, ‘Communication and Concurrency’, C. A. R. Hoare series ed., Prentice-Hall Int’l, 1989
- [15] J. von Neumann, ‘Theory of Self-Reproducing Automata’, edited and completed by A. W. Burks, Univ. of Illinois Press, Urbana, 1966
- [16] J. C. Reynolds, ‘Theories of Programming Languages’, Cambridge Univ. Press, 1998
- [17] Ravi Sethi, ‘Programming Languages: Concepts & Constructs’, 2nd ed., Addison-Wesley, 1996
- [18] K. Sutner, ‘Computation theory of cellular automata’, MFCS98 Satellite Workshop CA, Brno, Czech Rep., 1998
- [19] P. Tomic, G. Agha, ‘Complete characterization of phase spaces of certain types of threshold cellular automata’ (in preparation)
- [20] S. Wolfram ‘Twenty problems in the theory of CA’, Physica Scripta 9, 1985
- [21] S. Wolfram (ed.), ‘Theory and applications of CA’, World Scientific, Singapore, 1986
- [22] S. Wolfram, ‘Cellular Automata and Complexity (collected papers)’, Addison-Wesley, 1994

An Instrumentation Technique for Online Analysis of Multithreaded Programs

Grigore Roșu and Koushik Sen

Department of Computer Science,

University of Illinois at Urbana-Champaign, USA

Email: {grosu,ksen}@uiuc.edu

Abstract

A formal analysis technique aiming at finding safety errors in multithreaded systems at runtime is investigated. An automatic code instrumentation procedure based on multithreaded vector clocks for generating the causal partial order on relevant state update events from a running multithreaded program is first presented. Then, by means of several examples, it is shown how this technique can be used in a formal testing environment, not only to detect, but especially to predict safety errors in multithreaded programs. The prediction process consists of rigorously analyzing other potential executions that are consistent with the causal partial order: some of these can be erroneous despite the fact that the particular observed execution is successful. The proposed technique has been implemented as part of a Java program analysis tool. A bytecode instrumentation package is used, so the Java source code of the tested programs is not necessary.

1. Introduction and Motivation

A major drawback of testing is its lack of coverage: if an error is not exposed by a particular test case then that error is not exposed. To ameliorate this problem, test-case generation techniques have been developed to generate those test cases that can reveal potential errors with high probability [8, 18, 26]. Based on experience with and on the success in practice of related techniques already implemented in JAVA PATHEXPLORER (JPAX) [12, 11] and its sub-system EAGLE [4], we have proposed in [23, 24] a complementary approach to testing, which we call “predictive runtime analysis” and can be intuitively described as follows.

Suppose that a multithreaded program has a subtle safety error. Like in testing, one executes the program on some carefully chosen input (test case) and suppose that, unfortunately, the error is not revealed during that particular execution; such an execution is called *successful* with respect to that bug. If one regards the execution of a program as a flat, sequential trace of events or states, like NASA’s JPAX system [12, 11], University of Pennsylvania’s JAVA-MAC [17], or Bell Labs’ PET [10], then there is not much left to do to find the error except to run another test case. However, by observing the execution trace in a smarter way, namely

as a causal dependency partial order on state updates, one can predict errors that can potentially occur in other possible runs of the multithreaded program.

The present work is an advance in *runtime verification* [13], a more scalable and complementary approach to the traditional formal verification methods such as theorem proving and model checking [6]. Our focus here is on multithreaded systems with shared variables. More precisely, we present a simple and effective algorithm that enables an external observer of an executing multithreaded program to detect and predict specification violations. The idea is to properly *instrument* the system before its execution, so that it will emit relevant events at runtime. No particular specification formalism is adopted in this paper, but examples are given using a temporal logic that we are currently considering in JAVA MULTIPATHEXPLORER (JMPAX) [23, 24], a tool for safety violation prediction in Java multithreaded programs which supports the presented technique.

In multithreaded programs, threads communicate via a set of shared variables. Some variable updates can causally depend on others. For example, if a thread writes a shared variable x and then another thread writes y due to a statement $y = x + 1$, then the update of y *causally depends* upon the update of x . Only read-write, write-read and write-write causalities are considered, because multiple consecutive reads of the same variable can be permuted without changing the actual computation. A state is a map assigning values to shared variables, and a specification consists of properties on these states. Some variables may be of no importance at all for an external observer. For example, consider an observer which monitors the property “if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false”. All the other variables except x , y and z are irrelevant for this observer (but they can clearly affect the causal partial ordering). To minimize the number of messages sent to the observer, we consider a subset of *relevant events* and the associated *relevant causality*.

We present an algorithm that, given an executing multithreaded program, generates appropriate messages to be sent to an external observer. The observer, in order to perform its more elaborated system analysis, extracts the state update information from such messages together with the

relevant causality partial order among the updates. This partial order abstracts the behavior of the running program and is called *multithreaded computation*. By allowing an observer to analyze multithreaded computations rather than just flat sequences of events, one gets the benefit of not only properly dealing with potential reordering of delivered messages (reporting global state accesses), but also of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling and can be hard, if not impossible, to find by just testing.

To be more precise, let us consider a real-life example where a runtime analysis tool supporting the proposed technique, such as JMPAX, would be able to predict a violation of a property from a single, successful execution of the program. However, like in the case of data-races, the chance of detecting this safety violation by monitoring only the actual run is very low. The example consists of a two threaded program to control the landing of an airplane. It has three variables `landing`, `approved`, and `radio`; their values are 1 when the *plane is landing*, *landing has been approved*, and *radio signal is live*, respectively, and 0 otherwise. The safety property to verify is “If the plane has started landing, then it is the case that landing has been approved and since the approval the radio signal has never been down.”

The code snippet for a naive implementation of this control program is shown in Fig. 1. It uses some dummy functions, `askLandingApproval` and `checkRadio`, which can be implemented properly in a real scenario. The program has a serious problem that cannot be detected easily from a single run. The problem is as follows. Suppose the plane has received approval for landing and just before it started landing the radio signal went off. In this situation, the plane must abort landing because the property was violated. But this situation will very rarely arise in an execution: namely, when `radio` is set to 0 between the approval of landing and the start of actual landing. So a tester or a simple observer will probably never expose this bug. However, note that even if the radio goes off *after* the landing has started, a case which is quite likely to be considered during testing but in which the property is *not* violated, JMPAX will still be able to construct a possible run (counterexample) in which radio goes off between landing and approval. In Section 4, among other examples, it is shown how JMPAX is able to predict two safety violations from a single successful execution of the program. The user will be given enough information (the entire counterexample execution) to understand the error and to correct it. In fact, this error is an artifact of a bad programming style and cannot be easily fixed - one needs to give a proper event-based implementation. This example shows the power of the proposed runtime verification technique as compared to the existing ones in JPAX and JAVA-MAC.

The main contribution of this paper is a detailed presen-

```

int landing = 0, approved = 0, radio = 1;
void thread1(){
    askLandingApproval();
    if(approved==1){
        print("Landing approved");
        landing = 1;
        print("Landing started");}
    else {print("Landing not approved");}
}
void askLandingApproval(){
    if(radio==0) approved = 0
    else approved = 1; }

void thread2(){
    while(radio){checkRadio();} }
void checkRadio(){
    possibly change value of radio;}
```

Figure 1. A buggy implementation of a flight controller.

tation of an instrumentation algorithm which plays a crucial role in extracting the causal partial order from one flat execution, and which is based on an appropriate notion of vector clock inspired from [9, 21], called *multithreaded vector clock (MVC)*. An MVC V_i is associated to each thread t_i , and two MVCs V_x^a (access) and V_x^w (write) are associated to each shared variable x . When a thread t_i processes event e , which can be an internal event or a shared variable read/write, the code in Fig. 2 is executed. We prove that \mathcal{A} correctly implements the relevant causal partial order, i.e., that for any two messages $\langle e, i, V \rangle$ and $\langle e', j, V' \rangle$ sent by \mathcal{A} , e and e' are relevant and e causally precedes e' iff $V[i] \leq V'[i]$. This algorithm can be implemented in several ways. In the case of Java, we prefer to implement it as an appropriate instrumentation procedure of code or bytecode, to execute \mathcal{A} whenever a shared variable is accessed. Another implementation could be to modify a JVM. Yet another one would be to enforce shared variable updates via library functions, which execute \mathcal{A} as well. All these can add significant delays to the normal execution of programs.

ALGORITHM \mathcal{A}

INPUT: event e generated by thread t_i

1. if e is relevant then
 $V_i[i] \leftarrow V_i[i] + 1$
2. if e is a read of a shared variable x then
 $V_i \leftarrow \max\{V_i, V_x^w\}$
 $V_x^a \leftarrow \max\{V_x^a, V_i\}$
3. if e is a write of a shared variable x then
 $V_x^w \leftarrow V_x^a \leftarrow V_i \leftarrow \max\{V_x^a, V_i\}$
4. if e is relevant then
send message $\langle e, i, V_i \rangle$ to observer

Figure 2. The vector clock instrumentation algorithm.

2. Multithreaded Systems

We consider multithreaded systems in which several threads communicate with each other via a set of shared variables. A crucial point is that some variable updates can causally depend on others. We will present an algorithm which, given an executing multithreaded program, generates appropriate messages to be sent to an external observer. The observer, in order to perform its analysis, extracts the state update information from such messages together with the causality partial order among the updates.

2.1. Multithreaded Executions

Given n threads t_1, t_2, \dots, t_n , a *multithreaded execution* is a sequence of events $e_1 e_2 \dots e_r$, each belonging to one of the n threads and having type *internal*, *read* or *write* of a shared variable. We use e_i^j to represent the j -th event generated by thread t_i since the start of its execution. When the thread or position of an event is not important we can refer to it generically, such as e , e' , etc.; we may write $e \in t_i$ when event e is generated by thread t_i . Let us fix an arbitrary but fixed multithreaded execution, say \mathcal{M} , and let S be the set of all shared variables. There is an immediate notion of *variable access precedence* for each shared variable $x \in S$: we say e *x-precedes* e' , written $e <_x e'$, if and only if e and e' are variable access events (reads or writes) to the same variable x , and e “happens before” e' , that is, e occurs before e' in \mathcal{M} . This “happens-before” relation can be realized in practice by keeping a counter for each shared variable, which is incremented at each variable access.

2.2. Causality and Multithreaded Computations

Let \mathcal{E} be the set of events occurring in \mathcal{M} and let \prec be the partial order on \mathcal{E} :

- $e_i^k \prec e_i^l$ if $k < l$;
- $e \prec e'$ if there is $x \in S$ with $e <_x e'$ and at least one of e, e' is a write;
- $e \prec e''$ if $e \prec e'$ and $e' \prec e''$.

We write $e \parallel e'$ if $e \not\prec e'$ and $e' \not\prec e$. The partial order \prec on \mathcal{E} defined above is called the *multithreaded computation* associated with the original multithreaded execution \mathcal{M} . Synchronization of threads can be easily and elegantly taken into consideration by just generating appropriate read/write events when synchronization objects are acquired/released, so the simple notion of multithreaded computation as defined above is as general as practically needed. A permutation of all events e_1, e_2, \dots, e_r that does not violate the multithreaded computation, in the sense that the order of events in the permutation is consistent with \prec , is called a *consistent multithreaded run*, or simply, a *multithreaded run*.

A multithreaded computation can be thought of as the *most general assumption* that an observer of the multithreaded execution can make about the system without

knowing its semantics. Indeed, an external observer simply *cannot disregard* the order in which the same variable is modified and used within the observed execution, because this order can be part of the intrinsic semantics of the multithreaded program. However, multiple causally independent modifications of different variable can be permuted, and the particular order observed in the given execution is not critical. By allowing an observer to analyze *multithreaded computations* rather than just *multithreaded executions* like JPAX [12, 11], JAVA-MAC [17], and PET [10], one gets the benefit of not only properly dealing with potential re-orderings of delivered messages (e.g., due to using multiple channels to reduce the monitoring overhead), but also of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling.

2.3. Relevant Causality

Some variables in S may be of no importance for an external observer. For example, consider an observer whose purpose is to check the property “if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false”; formally, using the interval temporal logic notation notation in [15], this can be compactly written as $(x > 0) \rightarrow [y = 0, y > z]$. All the other variables in S except x, y and z are essentially irrelevant for this observer. To minimize the number of messages, like in [20] which suggests a similar technique but for distributed systems in which reads and writes are not distinguished, we consider a subset $\mathcal{R} \subseteq \mathcal{E}$ of *relevant events* and define the \mathcal{R} -*relevant causality* on \mathcal{E} as the relation $\triangleleft := \prec \cap (\mathcal{R} \times \mathcal{R})$, so that $e \triangleleft e'$ if and only if $e, e' \in \mathcal{R}$ and $e \prec e'$. It is important to notice though that the other variables can also indirectly influence the relation \triangleleft , because they can influence the relation \prec .

3. Multithreaded Vector Clock Algorithm

In this section, inspired and stimulated by the elegance and naturality of vector clocks [9, 21, 3] in implementing causal dependency in distributed systems, we next devise an algorithm to implement the relevant causal dependency relation in multithreaded systems. Since in multithreaded systems communication is realized by shared variables rather than message passing, to avoid any confusion we call the corresponding vector-clock data-structures *multithreaded vector clocks* and abbreviate them (*MVC*). The algorithm presented next has been mathematically derived from its desired properties, after several unsuccessful attempts to design it on a less rigorous basis. In this section we present it also in a mathematically driven style, because we believe that it reflects an instructive methodology to devise instrumentation algorithms for multithreaded systems.

Let V_i be an n -dimensional vector of natural numbers for each $1 \leq i \leq n$. Since communication in multithreaded systems is done via shared variables, and since

reads and writes have different weights, we let V_x^a and V_x^w be two additional n -dimensional vectors for each shared variable x ; we call the former *access MVC* and the latter *write MVC*. All MVCs are initialized to 0. As usual, for two n -dimensional vectors, $V \leq V'$ iff $V[j] \leq V'[j]$ for all $1 \leq j \leq n$, and $V < V'$ iff $V \leq V'$ and there is some $1 \leq j \leq n$ such that $V[j] < V'[j]$; also, $\max\{V, V'\}$ is the vector with $\max\{V, V'\}[j] = \max\{V[j], V'[j]\}$ for each $1 \leq j \leq n$. Our goal is to find a procedure that updates these MVCs and emits a minimal amount of events to an external observer, which can further extract the relevant causal dependency relation. Formally, the requirements of such a procedure, say \mathcal{A} , which works as a filter of the given multithreaded execution, must include the following natural

Requirements for \mathcal{A} . After \mathcal{A} updates the MVCs as a consequence of the fact that thread t_i generates event e_i^k during the multithreaded execution \mathcal{M} , the following should hold:

- (a) $V_i[j]$ equals the number of relevant events of t_j that causally precede e_i^k ; if $j = i$ and e_i^k is relevant then this number also includes e_i^k ;
- (b) $V_x^a[j]$ equals the number of relevant events of t_j that causally precede the most recent event¹ that accessed (read or wrote) x ; if $i = j$ and e_i^k is a relevant read or write of x event then this number also includes e_i^k ;
- (c) $V_x^w[j]$ equals the number of relevant events of t_j that causally precede the most recent write event of x ; if $i = j$ and e_i^k is a relevant write of x then this number also includes e_i^k .

Finally and most importantly, \mathcal{A} should correctly implement the relative causality (stated formally in Theorem 3).

In order to derive our algorithm \mathcal{A} satisfying the properties above, let us first introduce some notation. For an event e_i^k of thread t_i , let $(e_i^k]$ be the indexed set $\{(e_i^k)_j\}_{1 \leq j \leq n}$, where $(e_i^k)_j$ is the set $\{e_j^l \mid e_j^l \in t_j, e_j^l \in \mathcal{R}, e_j^l \prec e_i^k\}$ when $j \neq i$ and the set $\{e_i^l \mid l \leq k, e_i^l \in \mathcal{R}\}$ when $j = i$.

Lemma 1 With the notation above, for $1 \leq j \leq n$:

- 1. $(e_j^l)_j \subseteq (e_j^{l'})_j$ if $l \leq l'$;
- 2. $(e_j^l)_j \cup (e_j^{l'})_j = (e_j^{\max\{l, l'\}})_j$ for any l and l' ;
- 3. $(e_j^l)_j \subseteq (e_i^k)_j$ for any $e_j^l \in (e_i^k)_j$; and
- 4. $(e_i^k)_j = (e_j^l)_j$ for some appropriate l .

Thus, by 4 above, one can uniquely and unambiguously encode a set $(e_i^k)_j$ by just a number, namely the size of the corresponding set $(e_j^l)_j$, i.e., the number of relevant events of thread t_j up to its l -th event. This suggests that if the MVC V_i maintained by \mathcal{A} stores that number in its j -th component then (a) in the list of requirements \mathcal{A} would be fulfilled.

Let us next move to the MVCs of reads and writes of shared variables. For a variable $x \in S$, let $a_x(e_i^k)$ and

$w_x(e_i^k)$ be, respectively, the most recent events that accessed x and wrote x in \mathcal{M} , respectively. If such events do not exist then we let $a_x(e_i^k)$ and/or $w_x(e_i^k)$ undefined; if e is undefined then we also assume that (e) is empty. We introduce the following notations for any $x \in S$:

$$(e_i^k)_x^a = \begin{cases} (e_i^k] & \text{if } e_i^k \text{ is an access to } x, \text{ and} \\ (a_x(e_i^k)] & \text{otherwise;} \end{cases}$$

$$(e_i^k)_x^w = \begin{cases} (e_i^k] & \text{if } e_i^k \text{ is a write to } x, \text{ and} \\ (w_x(e_i^k)] & \text{otherwise.} \end{cases}$$

Note that if \mathcal{A} is implemented such that V_x^a and V_x^w store the corresponding numbers of elements in the index sets of $(e_i^k)_x^a$ and $(e_i^k)_x^w$ immediately after event e_i^k is processed by thread t_i , respectively, then (b) and (c) in the list of requirements for \mathcal{A} are also fulfilled.

We next focus on how MVCs need to be updated by \mathcal{A} when event e_i^k is encountered. With the notation introduced, one can observe the following recursive properties, where $\{e_i^k\}_i^{\mathcal{R}}$ is the indexed set whose components are empty for all $j \neq i$ and whose i -th component is either the one element set $\{e_i^k\}$ when $e_i^k \in \mathcal{R}$ or the empty set otherwise:

Lemma 2 Given any event e_i^k in \mathcal{M} such that e_i^k is

- 1. An internal event then

$$\begin{aligned} (e_i^k] &= (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}}, \\ (e_i^k)_x^a &= (a_x(e_i^k)], \text{ for any } x \in S, \\ (e_i^k)_x^w &= (w_x(e_i^k)], \text{ for any } x \in S; \end{aligned}$$

- 2. A read of x event then

$$\begin{aligned} (e_i^k] &= (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (w_x(e_i^k)], \\ (e_i^k)_x^a &= (e_i^k] \cup (a_x(e_i^k)], \\ (e_i^k)_y^a &= (a_y(e_i^k)], \text{ for any } y \in S \text{ with } y \neq x, \\ (e_i^k)_y^w &= (w_y(e_i^k)], \text{ for any } y \in S; \end{aligned}$$

- 3. A write of x event then

$$\begin{aligned} (e_i^k] &= (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (a_x(e_i^k)], \\ (e_i^k)_x^a &= (e_i^k], \\ (e_i^k)_x^w &= (e_i^k], \\ (e_i^k)_y^a &= (a_y(e_i^k)], \text{ for any } y \in S \text{ with } y \neq x, \\ (e_i^k)_y^w &= (w_y(e_i^k)], \text{ for any } y \in S \text{ with } y \neq x. \end{aligned}$$

Since each component set of each of the indexed sets in these recurrences has the form $(e_j^l)_j$ for appropriate j and l , and since each $(e_j^l)_j$ can be safely encoded by its size, one can then safely encode each of the above indexed sets by an n -dimensional MVC; these MVCs are precisely V_i for all $1 \leq i \leq n$ and V_x^a and V_x^w for all $x \in S$. It is a simple exercise now to derive² the MVC update algorithm \mathcal{A} given in Section 1. Therefore, \mathcal{A} satisfies all the stated requirements (a), (b) and (c), so they can be used as properties next:

²An interesting observation here is that one can regard the problem of recursively calculating $(e_i^k]$ as a dynamic programming problem. As can often be done in dynamic programming problems, one can reuse space and derive the Algorithm \mathcal{A} .

¹Most recent with respect to the given multithreaded execution \mathcal{M} .

Theorem 3 If $\langle e, i, V \rangle$ and $\langle e', j, V' \rangle$ are two messages sent by \mathcal{A} , then $e \triangleleft e'$ if and only if $V[i] \leq V'[i]$ if and only if $V < V'$.

Proof: First, note that e and e' are both relevant. The case $i = j$ is trivial. Suppose $i \neq j$. Since, by requirement (a) for \mathcal{A} , $V[i]$ is the number of relevant events that t_i generated before and including e and since $V'[i]$ is the number of relevant events of t_i that causally precede e' , it is clear that $V[i] \leq V'[i]$ iff $e \prec e'$. For the second part, if $e \triangleleft e'$ then $V \leq V'$ follows again by requirement (a), because any event that causally precedes e also precedes e' . Since there are some indices i and j such that e was generated by t_i and e' by t_j , and since $e' \not\prec e$, by the first part of the theorem it follows that $V'[j] > V[j]$; therefore, $V < V'$. For the other implication, if $V < V'$ then $V[i] \leq V'[i]$, so the result follows by the first part of the theorem. \square

3.1. Synchronization and Shared Variables

Thread communication in multithreaded systems was considered so far to be accomplished by writing/reading shared variables, which were assumed to be known *a priori*. In the context of a language like Java, this assumption works only if the shared variables are declared *static*; it is less intuitive when synchronization and dynamically shared variables are considered as well. Here we show that, under proper instrumentation, the basic algorithm presented in the previous subsection also works in the context of synchronization statements and dynamically shared variables.

Since in Java synchronized blocks cannot be interleaved, so corresponding events cannot be permuted, locks are considered as shared variables and a write event is generated whenever a lock is acquired or released. This way, a causal dependency is generated between any exit and any entry of a synchronized block, namely the expected happens-before relation. Java synchronization statements are handled exactly the same way, that is, the shared variable associated to the synchronization object is written at the entrance and at the exit of the synchronized region. Condition synchronizations (wait/notify) can be handled similarly, by generating a write of a dummy shared variable by both the notifying thread before notification and by the notified thread after notification.

To handle variables that are dynamically shared, for each variable x of primitive type in each class the instrumentation program adds *access* and *write* MVCs, namely `_access_mvc_x` and `_write_mvc_x`, as new fields in the class. Moreover, for each read and write access of a variable of primitive type in any class, it adds codes to update the MVCs according to the multithreaded vector clock algorithm.

3.2. A Distributed Systems Interpretation

It is known that the various mechanisms for process interaction are essentially equivalent. This leads to the follow-

ing natural question: could it be possible to derive the MVC algorithm in this section from vector clock based algorithms implementing causality in distributed systems, such as the ones in [3, 7]. The answer to this question is: *almost*.

Since writes and accesses of shared variables have different impacts on the causal dependency relation, the most natural thing to do is to associate two processes to each shared variable x , one for accesses, say x^a and one for writes, say x^w . As shown in Fig. 3 right, a write of x by thread i can be seen as sending a “request” message to write x to the “access process” x^a , which further sends a “request” message to the “write process” x^w , which performs the action and then sends an acknowledgment message back to i . This is consistent with step 3 of the algorithm in Fig. 2; to see this, note that $V_x^w \leq V_x^a$ at any time.

However, a read of x is less obvious and does not seem to be interpretable by message passing updating the MVCs the standard way. The problem here is that the MVC of x^a needs to be updated with the MVC of the accessing thread i , the MVC of the accessing thread i needs to be updated with the current MVC of x^w in order to implant causal dependencies between previous writes of x and the current access, but the point here is that the MVC of x^w does *not* have to be updated by reads of x ; this is what allows reads to be permutable by the observer. In terms of message passing, like Fig. 3 shows, this says that the access process x^a sends a *hidden* request message to x^w (after receiving the read request from i), whose only role is to “ask” x^w send an acknowledgment message to i . By hidden message, marked with dotted line in Fig. 3, we mean a message which is not considered by the standard MVC update algorithm. The role of the acknowledgment message is to ensure that i updates its MVC with the one of the write access process x^w .

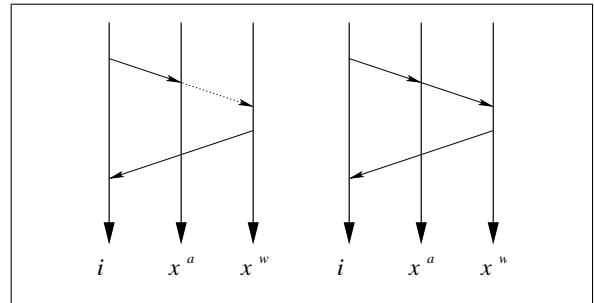


Figure 3. A distributed systems interpretation of reads (left) and writes (right).

4. The Vector Clock Algorithm at Work

In this section we propose predictive runtime analysis frameworks in which the presented MVC algorithm can be used, and describe by examples how we use it in JAVA MULTIPATHEXPLORER (JMPAX) [23, 24, 16].

The observer therefore receives messages of the form $\langle e, i, V \rangle$ in any order, and, thanks to Theorem 3, can ex-

tract the causal partial order \triangleleft on relevant events, which is its abstraction of the running program. Any permutation of the relevant events which is consistent with \triangleleft is called a *multithreaded run*, or simply a *run*. Notice that each run corresponds to some possible execution of the program under different execution speeds or scheduling of threads, and that the observed sequence of events is just one such run. Since each relevant event contains global state update information, each run generates a sequence of global states. If one puts all these sequences together then one gets a lattice, called *computation lattice*. The reader is assumed familiar with techniques on how to extract a computation lattice from a causal order given by means of vector clocks [21]. Given a global property to analyze, the task of the observer now is to verify it against every path in the automatically extracted computation lattice. JPAX and JAVA-MAC are able to analyze only one path in the lattice. The power of our technique consists of its ability to predict potential errors in other possible multithreaded runs.

Once a computation lattice containing all possible runs is extracted, one can start using standard techniques on debugging distributed systems, considering both state predicates [25, 7, 5] and more complex, such as temporal, properties [2, 5, 1, 4]. Also, the presented algorithm can be used as a front-end to partial order trace analyzers such as POTA [22]. Also, since the computation lattice acts like an abstract model of the running program, one can potentially run one's favorite model checker against any property of interest. We think, however, that one can do better than that if one takes advantage of the specific runtime setting of the proposed approach. The problem is that the computation lattice can grow quite large, in which case storing it might become a significant matter. Since events are received incrementally from the instrumented program, one can buffer them at the observer's side and then build the lattice on a level-by-level basis in a top-down manner, as the events become available. The observer's analysis process can also be performed incrementally, so that parts of the lattice which become non-relevant for the property to check can be garbage-collected while the analysis process continues.

If the property to be checked can be translated into a finite state machine (FSM) or if one can synthesize online monitors for it, like we did for safety properties [24, 14, 15, 23], then one can analyze all the multithreaded runs *in parallel*, as the computation lattice is built. The idea is to store the state of the FSM or of the synthesized monitor together with each global state in the computation lattice. This way, in any global state, all the information needed about the past can be stored via a set of states in the FSM or the monitor associated to the property to check, which is typically quite small in comparison to the computation lattice. Thus only one cut in the computation lattice is needed at any time, in particular one level, which significantly re-

duces the space required by the proposed predictive analysis algorithm.

Liveness properties apparently do not fit our runtime verification setting. However, stimulated by recent encouraging results in [19], we believe that it is also worth exploring techniques that can *predict violations of liveness properties*. The idea here is to search for paths of the form uv in the computation lattice with the property that the shared variable global state of the multithreaded program reached by u is the same as the one reached by uv , and then to check whether uv^ω satisfies the liveness property. The intuition here is that the system can potentially run into the infinite sequence of states uv^ω (u followed by infinity many repetitions of v), which may violate the liveness property. It is shown in [19] that the test $uv^\omega \models \varphi$ can be done in polynomial time and space in the sizes of u , v and φ , typically linear in uv , for almost any temporal logic.

4.1. Java MultiPathExplorer (JMPaX)

JMPaX [23, 24] is a runtime verification tool which checks a user defined specification against a running program. The specifications supported by JMPaX allow any temporal logic formula, using an interval-based notation built on state predicates, so our properties can refer to the entire history of states. Fig. 4 shows the architecture of JMPaX. An instrumentation module parses the user specifi-

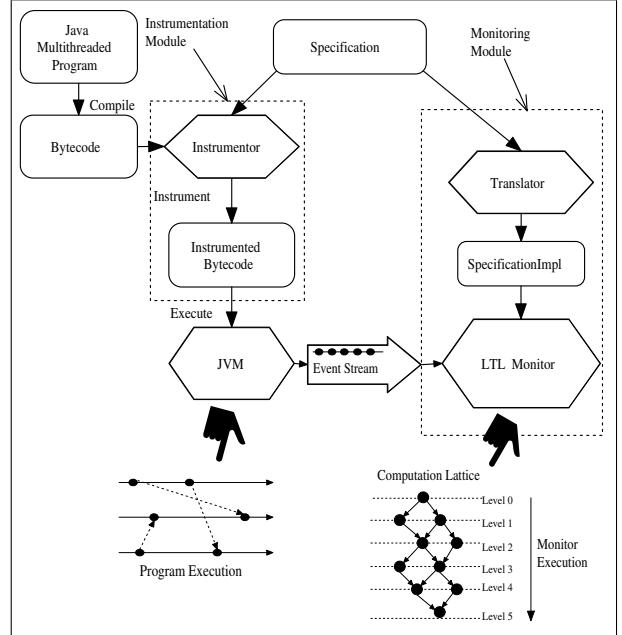


Figure 4. The Architecture of JMPaX.

tion, extracts the set of shared variables it refers to, i.e., the relevant variables, and then *instruments* the multithreaded program (which is assumed in bytecode form) as follows. Whenever a shared variable is accessed the MVC algorithm \mathcal{A} in Section 3 is inserted; if the shared variable is relevant

and the access is a write then the event is considered relevant. When the instrumented bytecode is executed, messages $\langle e, i, V \rangle$ for relevant events e are sent via a socket to an external observer.

The observer generates the computation lattice on a level by level basis, checking the user defined specification against all possible multithreaded runs in parallel. Note that only one of those runs was indeed executed by the instrumented multithreaded program, and that the observer does not know it; the other runs are *potential* runs, they can occur in other executions of the program. Despite the exponential number of potential runs, at most two consecutive levels in the computation lattice need to be stored at any moment. [23, 24] gives more details on the particular implementation of JMPAX. We next discuss two examples where JMPAX can predict safety violations from successful runs; the probability of detecting these bugs only by monitoring the observed run, as JPAX and JAVA-MAC do, is very low.

Example 1. Let us consider the simple landing controller in Fig.1, together with the property “If the plane has started landing, then it is the case that landing has been approved and since then the radio signal has never been down.” Suppose that a successful execution is observed, in which the radio goes down *after* the landing has started. After instrumentation, this execution emits only three events to the observer in this order: a write of approved to 1, a write of landing to 1, and a write of radio to 0. The observer can now build the lattice in Fig.5, in which the states are encoded by triples $\langle \text{landing}, \text{approved}, \text{radio} \rangle$ and the leftmost path corresponds to the observed execution. However, the lattice contains two other runs both violating the safety property. The rightmost one corresponds to the sit-

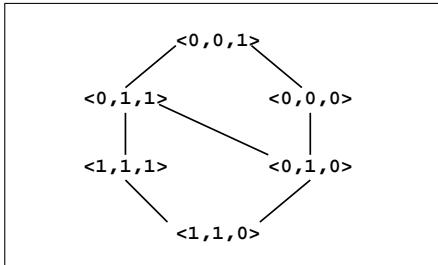


Figure 5. Computation lattice for the program in Fig. 1.

uation when the radio goes down right between the test `radio==0` and the action `approved=1`, and the inner one corresponds to that in which the radio goes down between the actions `approved=1` and `landing=1`. Both these erroneous behaviors are insightful and very hard to find by testing. JMPAX is able to build the two counterexamples very quickly, since there are only 6 states to analyze and three corresponding runs, so it is able to give useful feedback.

Example 2. Let us now consider an artificial example in-

tended to further clarify the prediction technique. Suppose that one wants to monitor the safety property “if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false” against a multithreaded program in which initially $x = -1$, $y = 0$ and $z = 0$, with one thread containing the code $x++$; ...; $y = x + 1$ and another containing $z = x + 1$; ...; $x++$. The dots indicate code that is not relevant, i.e., that does not access the variables x , y and z . This multithreaded program, after instrumentation, sends messages to JMPAX’s observer whenever the relevant variables x, y, z are updated. A possible execution of the program to be sent to the observer can consist of the sequence of program states $(-1, 0, 0)$, $(0, 0, 0)$, $(0, 0, 1)$, $(1, 0, 1)$, $(1, 1, 1)$, where the tuple $(-1, 0, 0)$ denotes the state in which $x = -1$, $y = 0$, $z = 0$. Following the MVC algorithm, we can deduce that the observer will receive the multithreaded computation shown in Fig. 6, which generates the computation lattice shown in the same figure. Notice that the observed multi-

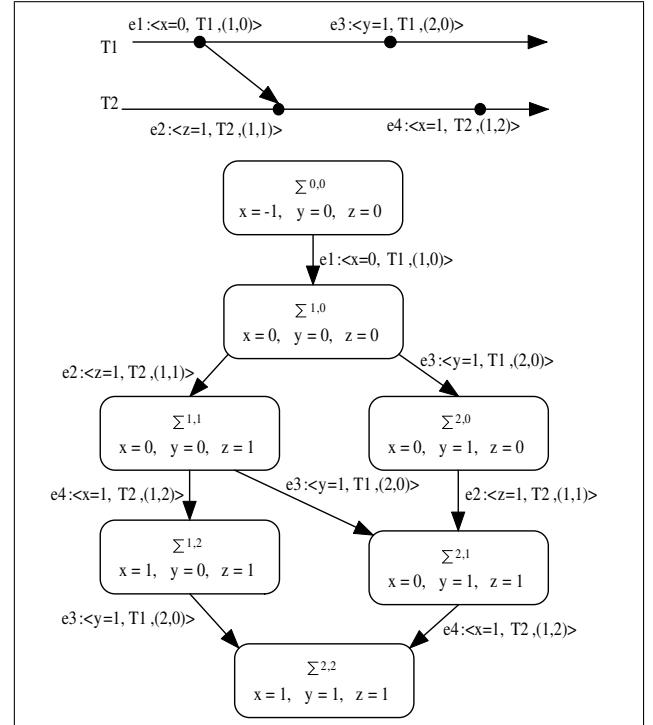


Figure 6. Computation lattice with three runs.

threaded execution corresponds to just one particular multithreaded run out of the three possible, namely the leftmost one. However, another possible run of the same computation is the rightmost one, which violates the safety property. Systems like JPAX and JAVA-MAC that analyze only the observed runs fail to detect this violation. JMPAX predicts this bug from the original successful run.

5. Conclusion

A simple and effective algorithm for extracting the relevant causal dependency relation from a running multithreaded program was presented in this paper. This algorithm is supported by JMPAX, a runtime verification tool able to detect and predict safety errors in multithreaded programs.

Acknowledgments. Many thanks to Gul Agha and Mark-Oliver Stehr for their inspiring suggestions and comments on several previous drafts of this work. The work is supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586, the DARPA IXO NEST Program, contract number F33615-01-C-1907), the ONR Grant N00014-02-1-0715, the Motorola Grant MOTOROLA RPS #23 ANT, and the joint NSF/NASA grant CCR-0234524.

References

- [1] M. Ahamad, M. Raynal, and G. Thia-Kime. An adaptive protocol for implementing causally consistent distributed services. In *Proceedings of International Conference on Distributed Computing (ICDCS'98)*, pages 86–93, 1998.
- [2] O. Babaoglu and M. Raynal. Specification and verification of dynamic properties in distributed computations. *Journal of Parallel and Distr. Computing*, 28(2):173–185, 1995.
- [3] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.
- [4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings Verification, Model Checking and Abstract Interpretation (VMCAI'04)* (To appear in LNCS), January 2004.
- [5] C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.
- [6] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.
- [7] R. Cooper and K. Marzullo. Consistent detection of global predicates. *ACM SIGPLAN Notices*, 26(12):167–174, 1991. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.
- [8] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of Formal Methods Europe (FME'93): Industrial Strength Formal Methods*, volume 670 of LNCS, pages 268–284, 1993.
- [9] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN/SIGOPS workshop on Parallel and Distr. Debugging*, pages 183–194. ACM, 1988.
- [10] E. L. Gunter, R. P. Kurshan, and D. Peled. PET: An interactive software testing tool. In *Computer Aided Verification (CAV'00)*, volume 1885 of LNCS, pages 552–556. Springer-Verlag, 2003.
- [11] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of ENTCS. Elsevier, 2001.
- [12] K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings Automated Software Engineering (ASE'01)*, pages 135–143. IEEE, 2001.
- [13] K. Havelund and G. Roşu. *Runtime Verification 2001, 2002*, volume 55, 70(4) of ENTCS. Elsevier, 2001, 2002. Proceedings of a Computer Aided Verification (CAV'01, CAV'02) workshop.
- [14] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools and Tech. Transfer*, to appear.
- [15] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of LNCS, pages 342–356. Springer, 2002.
- [16] Java MultiPathExplorer. <http://fsl.cs.uiuc.edu/jmpax>.
- [17] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of ENTCS. Elsevier Science, 2001.
- [18] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
- [19] N. Markey and P. Schnoebelen. Model checking a path (preliminary report). In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR'2003)*, LNCS. Springer, 2003.
- [20] K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WADG'91)*, volume 579 of LNCS, pages 254–272. Springer, 1991.
- [21] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1989.
- [22] A. Sen and V. K. Garg. Partial order trace analyzer (pota) for distributed programs. In *Proceedings of Workshop on Runtime Verification (RV'03)*, ENTCS, 2003.
- [23] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*. ACM, 2003.
- [24] K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)* (To Appear in LNCS), Barcelona, Spain, 2004. Springer.
- [25] S. D. Stoller. Detecting global predicates in distributed systems with clocks. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, pages 185–199, 1997.
- [26] S. A. Vilkomir and J. P. Bowen. Formalization of software testing criteria using the Z notation. In *Proceedings of COMPSAC 01: 25th IEEE Annual International Computer Software and Applications Conference*, pages 351–356. IEEE Computer Society, Oct. 2001.

Efficient Decentralized Monitoring of Safety in Distributed Systems

Koushik Sen, Abhay Vardhan, Gul Agha, Grigore Roşu

Department of Computer Science

University of Illinois at Urbana Champaign

{ksen, vardhan, agha, grosu}@cs.uiuc.edu

Abstract

We describe an efficient decentralized monitoring algorithm that monitors a distributed program's execution to check for violations of safety properties. The monitoring is based on formulae written in PT-DTL, a variant of past time linear temporal logic that we define. PT-DTL is suitable for expressing temporal properties of distributed systems. Specifically, the formulae of PT-DTL are relative to a particular process and are interpreted over a projection of the trace of global states that represents what that process is aware of. A formula relative to one process may refer to other processes' local states through remote expressions and remote formulae. In order to correctly evaluate remote expressions, we introduce the notion of KNOWLEDGE VECTOR and provide an algorithm which keeps a process aware of other processes' local states that can affect the validity of a monitored PT-DTL formula. Both the logic and the monitoring algorithm are illustrated through a number of examples. Finally, we describe our implementation of the algorithm in a tool called DIANA.

1. Introduction

Software errors from a number of different problems such as incorrect or incomplete specifications, coding errors, and faults and failures in the hardware, operating system or network. Model checking is an important technology which is finding increasing use as a means of reducing software errors. Unfortunately, despite impressive recent advances, the size of systems for which model checking is feasible remains rather limited. This weakness is particularly critical in the context of distributed systems: concurrency and asynchrony results in inherent non-determinism that significantly increases the number of states to be analyzed. As a result, most system builders must continue to use testing to identify bugs in their implementations.

There are two problems with software testing. First, testing is generally done in an *ad hoc* manner: the software developer must hand translate the requirements into specific dynamic checks on the program state. Second, test coverage

is often rather limited, covering only some execution paths. To mitigate the first problem, software often includes dynamic checks on a system's state in order to identify problems at run-time. Recently, there has been some interest in run-time monitoring techniques which provide a little more rigor in testing. In this approach, monitors are automatically synthesized from a formal specification. These monitors may then be deployed off-line for debugging or on-line for dynamically checking that safety properties are not being violated during system execution.

In this paper, we argue that distributed systems may be effectively monitored at runtime against formally specified safety requirements. By effective monitoring, we mean not only linear efficiency, but also decentralized monitoring where few or no additional messages need to be passed for monitoring purposes. We introduce an epistemic temporal logic for distributed knowledge. We illustrate the expressiveness of this logic by means of some simple examples. We then show how efficient distributed monitors may be synthesized from the specified requirements. Finally, we describe a distributed systems application development framework, called DIANA. To use DIANA, a user must provide an application together with the formal safety properties that she wants monitored. DIANA automatically synthesizes code for monitoring the specified requirements and weaves appropriate instrumentation code into the given application. The architecture of DIANA is illustrated in Figure 1.

The work presented in this paper was stimulated by the observation that in many distributed systems, such as wireless sensor networks, it is quite impractical to monitor requirements expressed in classical temporal logics. For example, consider a system of mobile nodes in which one mobile node may request a certain value from another mobile node. On receiving the request, the second node computes the value and returns it. An important requirement in such a system is that no node receives a reply from a node to which it has not previously issued a request. It is easy to see that Linear Temporal Logic (LTL) would not be a practical specification language for any reasonably sized collec-

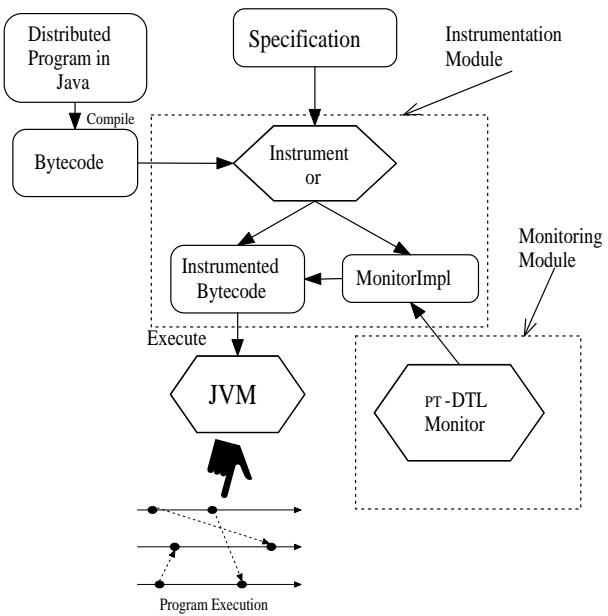


Figure 1. The Architecture of DIANA

tion of nodes. To use LTL, we would need to collect consistent snapshots of the global system; a monitor would then check the snapshots for possible violations of the property by considering all possible interleavings of events that are allowed by the distributed computation. In a system of thousands of nodes, collecting such a global snapshot would be prohibitive. Moreover, the number of possible interleavings to be considered would be large even if powerful techniques such as partial order reduction are used.

To address the above difficulty, we define *past-time distributed temporal logic* (PT-DTL). Using PT-DTL, one can check a property such as the one above by having a local monitor on each node. For example, node a monitors “if a has received a value then it must be the case that previously in the past at b the following held: b has computed the value and at a a request was made for that value in the past”. This is precisely and concisely expressed by the PT-DTL formula:

$$\text{receivedValue} \rightarrow \\ @_b(\Diamond(\text{computedValue} \wedge @_a(\Diamond\text{requestedValue})))$$

Note that we read $@$ as “at”, $@_b F$ is the value of F in the most recent local state of b that the current process is aware of, and \Diamond denotes the formula was true sometime in the past. Monitoring the above formula involves sending no additional messages – it involves inserting only a few bits of information which are piggybacked on the messages that are already being passed in the computation. This efficiency provides a substantial improvement over what is required to monitor formulas written in classical LTL.

We introduce *remote expressions* in PT-DTL to represent values which are functions depending on the state of a remote process. For example, a process may monitor the property: “if my alarm has been set then it must be the case that the difference between my temperature and the temperature at process b exceeded the allowed value”. This is expressed as:

$$\text{alarm} \rightarrow \Diamond((\text{myTemp} - @_b\text{otherTemp}) > \text{allowed})$$

Here $@_b\text{otherTemp}$ is a remote expression that is subtracted from the local value of myTemp .

An example of a safety property that may be useful in the context of an airplane software is: “if my airplane is landing then the runway allocated by the airport matches the one that I am planning to use”. This property may be expressed in PT-DTL as follows:

$$\text{landing} \rightarrow (\text{runway} = (@_\text{airport}\text{allocRunway}))$$

Many researchers have proposed temporal logics to reason about distributed systems. Most of these logics are inspired by the classic work of Aumann [5] and Halpern *et al.* [7] on knowledge in distributed systems. Meenakshi *et al.* define a knowledge temporal logic interpreted over a message sequence charts in a distributed system [16] and develop methods for model checking formulae in this logic. Our communication primitive was in part inspired by this work, but we allow arbitrary expressions and atomic propositions over expressions in their logic.

Another closely related work is that of Penczek [17, 18] which defines a temporal logic of causal knowledge. Knowledge operators are provided to reason about the local history of a process, as well as about the knowledge it acquires from other processes. However, in order to keep the complexity of model checking tractable, Penczek does not allow the nesting of causal knowledge operators. Interestingly, the nesting of causal knowledge operators does not add any complexity to our algorithm for monitoring.

Leucker investigates linear temporal logic interpreted over restricted labeled partial orders called Mazurkiewicz traces [12]. An overview of distributed linear time temporal logics based on Mazurkiewicz traces is given by Thiagarajan *et al.* in [22]. Alur *et al.* [4] introduce a temporal logic of causality (TLC) which is interpreted over causal structures corresponding to partial order executions of a distributed system. They use both past and future time operators and give a model checking algorithm for the logic.

In recent years, there has been considerable interest in runtime verification [1]. Havelund *et al.* [10] give algorithms for synthesizing efficient monitors for safety properties. Sen *et al.* [20] develop techniques for runtime safety analysis for multithreaded programs and introduce the tool JMPAX. Some other runtime verification systems include JPaX from NASA Ames [9] and UPENN’s Mac [11].

We can think of at least three major contributions of the work presented in this paper. First, we define a simple but expressive logic to specify safety properties in distributed systems. Second, we provide an algorithm to synthesize decentralized monitors for safety properties that are expressed in the logic. Finally, we describe the implementation of a tool (DIANA) that is based on this technique. The tool is publicly available for download.

The rest of the paper is organized as follows. Section 2 and Section 3 give the preliminaries. Section 4 introduces PT-DTL. In Section 5 we describe the algorithm that underlies our implementation. Section 6 briefly describes the implementation along with initial experimentation.

2. Distributed Systems

A distributed system is a collection of n processes or actors (p_1, \dots, p_n), each with its own local state. The local state of a process is given by the values bound to its variables. Note that there are no global or shared variables. Processes communicate with other using asynchronous messages whose order of arrival is indeterminate. The computation of each process is abstractly modeled by a set of *events*, and a distributed computation is specified by a partial order \prec on the events. There are three types of events:

1. *internal* events change the local state of a process;
2. *send* events cause a process to send a message;
3. *receive* events occur when a message is received by a process.

Let E_i denote the set of events of process p_i and let E denote $\bigcup_i E_i$. Now, $\prec \subseteq E \times E$ is defined as follows:

1. $e \prec e'$ if e and e' are events of the same process and e happens immediately before e' ,
2. $e \prec e'$ if e is the send event of a message at some process and e' is the corresponding receive event of the message at the recipient process.

The partial order \prec is the transitive closure of the relation \lessdot . This partial order captures the *causality* relation between events. The structure described by $\mathcal{C} = (E, \prec)$ is called a *distributed computation* and we assume an arbitrary but given distributed computation \mathcal{C} . Further, \preccurlyeq is the reflexive and transitive closure of \lessdot . In Fig. 2, $e_{11} \prec e_{23}$, $e_{12} \prec e_{23}$, and $e_{11} \preccurlyeq e_{23}$. However, $e_{12} \not\preccurlyeq e_{23}$.

For $e \in E$, we define $\downarrow e \stackrel{\text{def}}{=} \{e' \mid e' \preccurlyeq e\}$, that is, $\downarrow e$ is the set of events that causally precede e . For $e \in E_i$, we can think of $\downarrow e$ as the local state of p_i when the event e has just occurred. This state contains the history of events of all processes that causally precede e .

We extend the definition of \lessdot , \prec and \preccurlyeq to local states such that $\downarrow e \lessdot \downarrow e'$ iff $e \lessdot e'$, $\downarrow e \prec \downarrow e'$ iff $e \prec e'$, and $\downarrow e \preccurlyeq \downarrow e'$ iff $e \preccurlyeq e'$. We denote the set of local states of a process

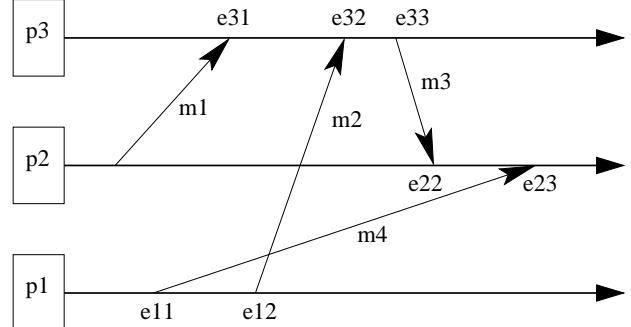


Figure 2. Sample Distributed Computation

p_i by $LS_i \stackrel{\text{def}}{=} \{\downarrow e \mid e \in E_i\}$ and let $LS \stackrel{\text{def}}{=} \bigcup_i LS_i$. We use the symbols s_i, s'_i, s''_i, \dots to represent the local states of process p_i . We also assume that the local state s_i of each process p_i associates values to some local variables V_i , and that $s_i(v)$ denotes the value of a variable $v \in V_i$ in the local state s_i at process p_i .

We use the notation $causal_j(s_i)$ to refer to the latest state of process p_j that the process p_i knows while in state s_i . Formally, if $causal_j(s_i) = s_j$ then $s_j \in LS_j$ and $s_j \preccurlyeq s_i$ and for all $s'_j \in LS_j$ if $s'_j \preccurlyeq s_i$ then $s'_j \preccurlyeq s_j$. For example, in Figure 2 $causal_1(\downarrow e_{23}) = \downarrow e_{12}$. Note that if $i = j$ then $causal_j(s_i) = s_i$.

3. Past Time Linear Temporal Logic (PT-LTL)

Past-time Linear Temporal Logic (PT-LTL) [13, 14] has been used in [10, 11, 20] to express, monitor and predict violations of safety properties of software systems. The syntax of PT-LTL is as follows:

$$F ::= \begin{array}{l} \text{true} \mid \text{false} \mid a \in A \mid \neg F \mid F \text{ op } F \quad \text{propositional} \\ \mid \odot F \mid \Diamond F \mid \Box F \mid F \mathcal{S} F \quad \text{temporal} \end{array}$$

where op are standard binary operators, $\wedge, \vee, \rightarrow$, and \leftrightarrow . $\odot F$ should be read as “previously”, $\Diamond F$ as “eventually in the past”, $\Box F$ as “always in the past”, $F_1 \mathcal{S} F_2$ as “ F_1 since F_2 ”.

The logic is interpreted on a finite sequence of states or a *run*. If $\rho = s_1 s_2 \dots s_n$ is a run then we let ρ_i denote the prefix run $s_1 s_2 \dots s_i$ for each $1 \leq i \leq n$. The semantics of the different operators is given in Table 1.

For example, the formula $\Box((action \wedge \odot \neg action) \rightarrow (\neg stop \mathcal{S} start))$ states that whenever *action* starts to be true, it is the case that *start* was true at some point in the past and since then *stop* was never true: in other words, the action is taken only while the system is active.

Notice that the semantics of “previously” is given as if the trace is unbounded in the past and stationary in the first event. In runtime monitoring, we start the process of monitoring from the point that the first event is generated and we continue monitoring for as long as events are generated.

$\rho \models \text{true}$	for all ρ ,
$\rho \not\models \text{false}$	for all ρ ,
$\rho \models a$	iff a holds in the state s_n ,
$\rho \models \neg F$	iff $\rho \not\models F$,
$\rho \models F_1 op F_2$	iff $\rho \models F_1$ and/or/implies/iff $\rho \models F_2$, when op is $\wedge / \vee / \rightarrow / \leftrightarrow$,
$\rho \models \odot F$	iff $\rho' \models F$, where $\rho' = \rho_{n-1}$ if $n > 1$ and $\rho' = \rho$ if $n = 1$,
$\rho \models \Diamond F$	iff $\rho_i \models F$ for some $1 \leq i \leq n$,
$\rho \models \Box F$	iff $\rho_i \models F$ for all $1 \leq i \leq n$,
$\rho \models F_1 \mathcal{S} F_2$	iff $\rho_j \models F_2$ for some $1 \leq j \leq n$ and $\rho_i \models F_1$ for all $j < i \leq n$,

Table 1. Semantics of PT-LTL

Although PT-LTL is interpreted over a linear execution trace, in distributed systems a computation is a partial order which may have several possible linearizations. Therefore, monitoring a distributed computation requires monitoring all possible linear traces that may be obtained from a partial order. Unfortunately, the number of linearizations of a partial order may be exponential in the length of the computation and thus monitoring PT-LTL formula may be intractable. A major contribution of this paper is to extend PT-LTL so that we can reason about a distributed property using only local monitoring. We describe this extension next.

4. Past Time Distributed Temporal Logic

Although PT-LTL works well for a single process, once we have more processes interacting with each other we need to reason about the state of remote processes. Since practical distributed systems are usually asynchronous and the absolute global state of the system is *not* available to processes, the best thing that each process can do it to reason about the global state that it is *aware of*.

We define Past-Time Distributed Temporal Logic (PT-DTL) by extending PT-LTL to express safety properties of distributed message passing systems. Specifically, we add a pair of *epistemic operators* as in [19], written $@$, whose role is to evaluate an expression or a formula in the *last known state* of a remote process. We call such an expression or a formula *remote*. A remote expression or formula may contain nested epistemic operators and refer to variables that are local to a remote process. By using remote expressions, in addition to remote formulae, a larger class of desirable properties of distributed systems may be specified without sacrificing the efficiency of monitoring.

For example, consider the simple local property at a process p_i that if α is true in the current local state of p_i then β must be true at the latest state of process p_j of which p_i is aware of. This property will be written formally in PT-DTL as $\alpha \rightarrow @_j \beta$. However, referring to remote formulae only is *not* sufficient to express a broad range of useful global properties such as “at process p_i , the value of x in the current

state is greater than the value of y at process p_j in the latest causally preceding state.” The reason we introduce the novel epistemic operators on expressions is that it is crucial to be able to also refer to *values* of expressions in remote local states. For example, the property above can be formally specified as the PT-DTL formula $x > @_j y$ at process p_i where $@_j y$ is the value of y at process p_j that p_i is aware of.

The intuition underlying PT-DTL is that each process is associated with local temporal formulae which may refer to the global state of the distributed system. These formulae are required to be valid at the respective processes during a distributed computation. A distributed computation satisfies the specification when all the local formulae are shown to satisfy the computation.

4.1. Syntax

From now on, we will use PT-DTL formula only in the context of a particular process, say p_i . We call such formulae *i-formulae* and denote them as F_i, F'_i, \dots . Moreover, we introduce *i-expressions*, expressions that are local to a process p_i , and denoted them by ξ_i, ξ'_i, \dots . Informally, an *i-expression* is an expression over the global state of the system that process p_i is currently aware of. Local predicates on *i-expressions* form the atomic propositions on which the temporal *i-formulae* are built.

We add the *epistemic operators* $@_j$ that take j -expressions or j -formulae and convert them into expressions or formulae local to process p_i . Informally, $@_j$ yields an expression or a formula on process p_j over the projection of the global state that the current process is aware of. The following gives the formal syntax of PT-DTL with respect to a process p_i , where i and j are any process indices (not necessarily distinct):

$$\begin{aligned}
F_i ::= & \text{ true } | \text{ false } | P(\vec{\xi}_i) | \neg F_i | F_i \text{ op } F_i & \text{ propositional} \\
& | \odot F_i | \Diamond F_i | \Box F_i | F_i \mathcal{S} F_i & \text{ temporal} \\
& | @_j F_j & \text{ epistemic} \\
\xi_i ::= & c | v_i | f(\vec{\xi}_i) & \text{ functional} \\
& | @_j \xi_j & \text{ epistemic} \\
\vec{\xi}_i ::= & (\xi_i, \dots, \xi_i)
\end{aligned}$$

The infix operator op may be a binary propositional operator such as $\wedge, \vee, \rightarrow$ or \equiv . The term $\vec{\xi}_i$ stands for a tuple of expressions on process p_i . The term $P(\vec{\xi}_i)$ is a (computable) predicate over the tuple $\vec{\xi}_i$ and $f(\vec{\xi}_i)$ is a (computable) function over the tuple. For example, P may be $<, \leq, >, \geq, =$ and f may be $+, -, /, *$. Variables v_i belongs to the set V_i which contains all the local state variables of process p_i . Constants such as $0, 1, 3.4$ are represented by c, c', c_1, \dots

The expression $@_j\xi_j$ is an i -expression representing the remote expression ξ_j . Similarly, $@_jF_j$ is an i -formula referring to the local knowledge about the remote validity of j -formula F_j . In other words, $@_j$ converts a j -expression or a j -formula to an i -expression or an i -formula, respectively.

4.2. Semantics

The semantics of PT-DTL is a natural extension of PT-LTL with the expected behavior for the epistemic operators. The atomic propositions of PT-LTL are replaced by predicates over tuples of expressions. Table 2 formally gives the semantics of each operator of PT-DTL. $(C, s_i)[@_j\xi_j]$ is the value of the expression ξ_j in the state $s_j = causal(s_i)$ which is the latest state of process p_j of which process p_i is aware of. We assume that expressions are properly typed. Typically, these types could be: integer, real, strings. We assume that $s_i, s'_i, s''_i, \dots \in LS_i$ and $s_j, s'_j, s''_j, \dots \in LS_j$. Notice that, as in PT-LTL, the meaning of the “previously” operator on the initial state of each process reflects the intuition that the execution trace is unbounded in the past and *stationary*. We consider this as the most reasonable assumption that one can make about the past.

4.3. Examples

To illustrate our logic, we consider a few relatively standard examples in the distributed systems literature (see, e.g., [21]). The first example is *leader election* for a network of processes. The key requirement for leader election is that there is at-most one leader. Assume the number of processes is n , and *state* is a variable in each process that can have values *leader*, *loser*, *candidate*, *sleep*. We can formulate the key leader election property at every process as: “if a leader is elected then if the current process is a leader then, to its knowledge, none of the other processes is a leader” written as the PT-DTL i -local formula:

$$\text{leaderElected} \rightarrow (\text{state} = \text{leader} \rightarrow \bigwedge_{j \neq i} (@_j(\text{state} \neq \text{leader}))$$

Given an implementation of the leader election problem, one can monitor this formula at each process. If the prop-

erty is violated, then clearly the leader election implementation is incorrect.

The second example is *majority vote*. The desired property, “if the resolution is accepted then more than half of the processes say yes”, can be stated as:

$$\text{accepted} \rightarrow (@_1(\text{vote}) + @_2(\text{vote}) + \dots + @_n(\text{vote})) > n/2$$

where, a process stores 1 in a local variable *vote* if it is in favor of the resolution, and 0 otherwise.

A third example is a safety property that a server must satisfy in case it reboots itself: “the server accepts the command to reboot only after knowing that each client is inactive and aware of the warning about pending reboot.” The property is expressed as the *server*-local formula below which contains nested epistemic operators:

$$\text{rebootAccepted} \rightarrow \bigwedge_{\text{client}} (@_{\text{client}}(\text{inactive} \wedge @_{{\text{server}}} \text{rebootWarning}))$$

5. Monitoring Algorithm for PT-DTL

We describe an automated technique to synthesize efficient distributed monitors for safety properties in distributed systems expressed in PT-DTL. We assume that one or more processes are associated with PT-DTL formulae that must be satisfied by the distributed computation. The synthesized monitor is *distributed*, in the sense that it consists of separate, *local monitors* running on each process. A local monitor may attach additional information to an outgoing message from the corresponding process. This information can subsequently be extracted by the monitor on the receiving side without changing the underlying semantics of the distributed program. The key guiding principles in the design of this technique are:

- A local monitor should be fast, so that monitoring can be done online;
- A local monitors should have little memory overhead, in particular, it should *not* need to store the entire history of events on a process; and
- The number of messages that need to be sent between processes for the purpose of monitoring should be minimal.

In this section, when we refer to a remote expression or formulae we mean an expression which occurs in any of the monitored PT-DTL formulae.

5.1. Knowledge Vectors

Consider the problem of evaluating a remote j -expression $@_j\xi_j$ at process p_i . A naive solution is that process p_j simply piggybacks the value of ξ_j evaluated at p_j ,

$\mathcal{C}, s_i \models \text{true}$	for all s_i
$\mathcal{C}, s_i \not\models \text{false}$	for all s_i
$\mathcal{C}, s_i \models P(\xi_i, \dots, \xi'_i)$	iff $P((\mathcal{C}, s_i)[\xi_i], \dots, (\mathcal{C}, s_i)[\xi'_i]) = \text{true}$
$\mathcal{C}, s_i \models \neg F_i$	iff $\mathcal{C}, s_i \not\models F_i$
$\mathcal{C}, s_i \models F_i \text{ op } F'_i$	iff $\mathcal{C}, s_i \models F_i \text{ op } \mathcal{C}, s_i \models F'_i$
$\mathcal{C}, s_i \models \odot F_i$	iff if $\exists s'_i . s'_i \preceq s_i$ then $\mathcal{C}, s'_i \models F_i$ else $\mathcal{C}, s_i \models F_i$
$\mathcal{C}, s_i \models \diamond F_i$	iff $\exists s'_i . s'_i \preceq s_i$ and $\mathcal{C}, s'_i \models F_i$
$\mathcal{C}, s_i \models \square F_i$	iff $\mathcal{C}, s_i \models F_i$ for all $s'_i \preceq s_i$
$\mathcal{C}, s_i \models F_i \mathcal{S} F'_i$	if $\exists s'_i . s'_i \preceq s_i$ and $\mathcal{C}, s'_i \models F'_i$ and $\forall s''_i . s'_i \prec s''_i \preceq s_i$ implies $\mathcal{C}, s''_i \models F_i$
$\mathcal{C}, s_i \models @_j F_j$	iff $\mathcal{C}, s_j \models F_j$ where $s_j = \text{causal}(s_i)$
$(\mathcal{C}, s_i)[v_i]$	$= s_i(v_i)$, that is, the value of v_i in s_i
$(\mathcal{C}, s_i)[c_i]$	$= c_i$
$(\mathcal{C}, s_i)[f(\xi_i, \dots, \xi'_i)]$	$= f((\mathcal{C}, s_i)[\xi_i], \dots, (\mathcal{C}, s_i)[\xi'_i])$
$(\mathcal{C}, s_i)[@_j \xi_j]$	$= (\mathcal{C}, s_j)[\xi_j]$ where $s_j = \text{causal}_j(s_i)$

Table 2. Semantics of PT-DTL

with every message that it sends out. The recipient process p_j can extract this value and use it as the value of $@_j \xi_j$. However, this approach is problematic: recall that messages from p_j could reach p_i in an arbitrary order: because the arrival order of two messages, even from the same sender, is indeterminate, more recent values may be overwritten by older ones. To keep track of the causal history, or in other words the most recent knowledge, we add an event number corresponding to the local history sequence at p_j at the time expressions were sent out in messages. Stale information in a reordered message sequence is then simply discarded.

Causal ordering can be effectively accomplished by using an array called KNOWLEDGEVECTOR with an entry for any process p_j for which there is an occurrence of $@_j$ in any PT-DTL formula at any process. Note that knowledge vectors are motivated and inspired by vector clocks [8, 15]. The size of KNOWLEDGEVECTOR is not dependent on the number of processes but on the number of remote expressions and formulae. Let $KV[j]$ denote the entry for process p_j on a vector KV . $KV[j]$ contains the following fields:

- The sequence number of the last event seen at p_j , denoted by $KV[j].seq$;
- A set of values $KV[j].values$ storing the values j -expressions and j -formulae.

Each process p_i keeps a local KNOWLEDGEVECTOR denoted by KV_i . The monitor of process p_i attaches a copy of KV_i with every outgoing message m . We denote the copy by KV_m . The algorithm for the update of KNOWLEDGEVECTOR KV_i at process p_i is as follows:

1. **[internal]:** update $KV_i[i]$. Evaluate $eval(\xi_i, s_i)$ and $eval(F_i, s_i)$ (see Subsection 5.2) for each $@_i \xi_i$

and $@_i F_i$, respectively, and store them in the set $KV_i[i].values$;

2. **[send m]:** $KV_i[i].seq \leftarrow KV_i[i].seq + 1$. Send KV_i with m as KV_m ;
3. **[receive m]:** for all j , if $KV_m[j].seq > KV_i[j].seq$ then $KV_i[j] \leftarrow KV_m[j]$, that is,
 $KV_i[j].seq \leftarrow KV_m[j].seq$, and
 $KV_i[j].values \leftarrow KV_m[j].values$.

We call this the KNOWLEDGEVECTOR algorithm. Informally, $KV_i[j].values$ contains the latest values that p_i has for j -expressions or j -formulae. Therefore, for the value of a remote expression or formula of the form $@_j \xi_j$ or $@_j F_j$ in the set $KV_i[j].values$. Note that the sequence number needs to be incremented only when sending messages. The correctness of the algorithm is relatively straightforward and we skip its formal proof.

Proposition 1 *For any process p_i and any j , the entry for ξ_j or F_j in $KV_i[j].values$ contains the value of $@_j \xi_j$ or $@_j F_j$, respectively.*

The above algorithm tries to minimize the local work when sending a message. However, observe that the values calculated at step 1 are needed only when an outgoing message is generated at step 2, so one could have just evaluated all the expressions ξ_i and F_i at step 2, right before the message is sent out. This would reduce the runtime overhead at step 1 but it would increase it at step 2. For different applications, different alternates may be more efficient.

The initial values for all the variables in a distributed program can be found either by a static analysis of the program or by a distributed broadcast at the beginning of the

computation. Thus, it is assumed that each process p_i has the complete knowledge of the initial values of remote expressions for all processes. These values are used to initialize the entries $KV_i[j].values$ in the KNOWLEDGEVECTOR of p_i for all j .

5.2. Monitoring a Local PT-DTL Formula

The monitoring algorithm for a PT-DTL formula is similar in spirit to that for an ordinary PT-LTL formula described in [20]. The key difference is that we allow remote expressions and remote formulae whose values and validity, respectively, need to be transferred from the remote process to the current process. Once the KNOWLEDGEVECTOR is properly updated, the local monitor can compute the boolean value of the formula to be monitored, by recursively evaluating the boolean value of each of its subformulae in the current state. To do so, it may also use the boolean values of subformulae evaluated in the previous state and the values of remote expressions and remote formulae.

A function $eval$ is defined next. $eval$ takes advantage of the recursive nature of the temporal operators (see Table 3) to calculate the boolean value of a formula in the current state in terms of (a) its boolean value in the previous state and (b) the boolean value of its subformulae in the current state. The function $op(F_i)$ returns the operator of the formula F_i , $binary(op(F_i))$ returns *true* if $op(F_i)$ is binary, $unary(op(F_i))$ returns *true* if $op(F_i)$ is unary, $left(F_i)$ returns the left subformula of F_i , $right(F_i)$ returns the right subformula of F_i when $op(F_i)$ is binary, and $subformula(F_i)$ returns the subformula of F_i otherwise. The variable $index$ represents the index of a subformula.

```

array now; array pre; int index;

boolean eval(Formula Fi, State si){
    if binary(op(Fi)) then{
        lval ← eval(left(Fi), si);
        rval ← eval(right(Fi), si); }
    else if unary(op(Fi)) then
        val ← eval(subformula(Fi), si);
        index ← 0;
    case(op(Fi)) of{
        true : return true; false : return false;
        P(ξ̄i) : return P(eval(ξi, si), ..., eval(ξ'i, si));
        op : return rval op lval; ¬: return not val;
        S : now[index] ← (pre[index] and lval) or rval;
            return now[index++];
        □, ◊, ⊕ : now[index] ← val; return now[index++];
        ◊ : now[index] ← pre[index] or val;
            return now[index++];
        ⊕ : now[index] ← val; return pre[index++];
        @jFj : return value of Fj from KVi[j].values;
    }
}

```

where, the global array pre contains the boolean values of all subformulae in the previous state that will be required in the current state, while the global array now , after the evaluation of $eval$, will contain the boolean values of all subformulae in the current state that may be required in the next state. Note that the now array's value is set in the function $eval$. The function $eval$ on expressions is defined next:

```

value eval(Expression ξi, State si){
    case(ξi) of{
        vi: return si(vi); ci: return ci;
        f(ξi1, ..., ξik): return f(eval(ξi1, si), ..., eval(ξik, si));
        @jξj: return value of ξj from KVi[j].values;
    }
}

```

Note that the function $eval$ cannot be used to evaluate the boolean value of a formula at the first event, as the recursion handles the case $n = 1$ in a different way. We define the function $init$ to handle this special case as implied by the semantics of PT-DTL in Tables 2 and 3 on one event traces:

```

boolean init(Formula Fi, State si){

    if binary(op(Fi)) then{
        lval ← init(left(Fi, si));
        rval ← init(right(Fi, si)); }
    else if unary(op(Fi)) then
        val ← init(subformula(Fi, si));
        index ← 0;
    case(op(Fi)) of{
        true : return true; false : return false;
        P(ξ̄i) : return P(eval(ξi, si), ..., eval(ξ'i, si));
        op : return rval op lval; ¬: return not val;
        S : now[index] ← rval; return now[index++];
        □, ◊, ⊕ : now[index] ← val; return now[index++];
    }
}

```

As mentioned earlier, in order to properly update the set $KV_i[i].values$, we can either use the function $eval$ after every internal event, or use it immediately before sending any message. If a monitored PT-DTL formula F_i is specified for a process p_i , we call p_i as the owner of that formula. At the owner process, we evaluate F_i using the $eval$ function after every internal and receive event and assign now to pre . This is done after the KNOWLEDGEVECTOR is updated, correspondingly after the event. If the evaluation of F_i is false then we report a warning that the formula F_i has been violated. The time and space complexity of this algorithm at every event is $\Theta(m)$, where m is the size of the original local formula.

5.3. Example

Consider three processes, p_1 , p_2 and p_3 . p_1 has a local variable x whose initial value is 5, p_2 has a local variable

$$\begin{aligned}
\mathcal{C}, s_i \models \diamond F_i &= \mathcal{C}, s_i \models F_i \text{ or } (\exists s'_i . s'_i \ll s_i \text{ and } \mathcal{C}, s'_i \models \diamond F_i) \\
\mathcal{C}, s_i \models \Box F_i &= \mathcal{C}, s_i \models F_i \text{ and } (\exists s'_i . s'_i \ll s_i \text{ implies } \mathcal{C}, s'_i \models \Box F_i) \\
\mathcal{C}, s_i \models F_i SF'_i &= \mathcal{C}, s_i \models F'_i \text{ or} \\
&\quad (\mathcal{C}, s_i \models F_i \text{ and } \exists s'_i . s'_i \ll s_i \text{ and } \mathcal{C}, s'_i \models F_i SF'_i)
\end{aligned}$$

Table 3. Recursive Semantics of PT-DTL

y with initial value 7 and p_2 monitors the formula $\Box(y \geq @_1x)$. An example computation is shown in Figure 3.

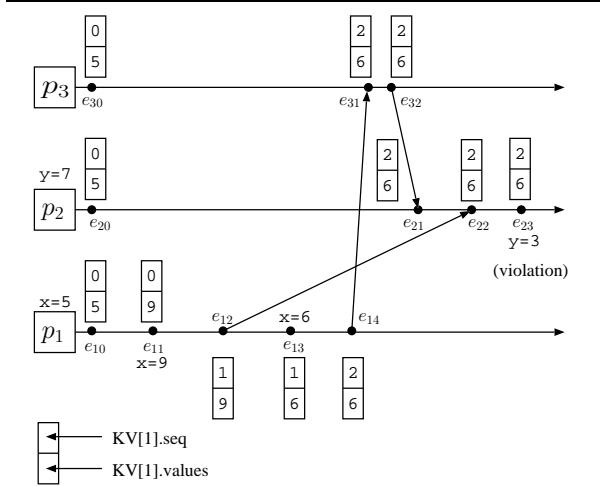


Figure 3. Monitoring of $\Box(y \geq @_1x)$ at p_2

There is only one formula to monitor with a single occurrence of an $@$ operator, namely $@_1x$. Hence, the KNOWLEDGEVECTOR has a single entry which corresponds to p_1 . Moreover, since the only remote expression to be tracked is x , $KV[1].values$ simply stores the value of x . In the figure, next to each event, we show $KV[1]$ at that instant for that process. $KV[1]$ is graphically displayed by a stack of two numbers, the top number showing $KV[1].seq$ and the bottom number showing the value for x .

The computation starts off with the initial values of $x = 5$ and $y = 7$. All processes know the initial value of x , hence the $KV[1].values$ for each process has value 5. It is easy to see that the monitored formula $\Box(y \geq @_1x)$ holds initially at p_2 . Subsequently, at p_1 there is an internal event e_{11} which sets $x = 9$ and updates $KV_1[1].values$ correspondingly. Process p_1 then sends a message to p_2 with a copy of its current KV . Another internal event e_{13} causes x to be set to 6. Process p_1 again sends a message, this time to p_3 , with the updated KV . Process p_3 updates its KV and sends this on the message it sends to p_2 .

At process p_2 , the message sent by p_3 happens to arrive earlier than the message from p_1 . Therefore, at event e_{21} ,

on receiving the message from p_3 , process p_2 is able to update its KV to the one sent at event e_{14} . The monitor at p_2 again evaluates the property and finds that it still holds. The message sent by p_1 finally arrives at e_{22} but the KV piggybacked on is ignored as it has a smaller $KV[1].seq$ than $KV_2[1].seq$. The monitor correctly continues to declare the property valid. However, another internal event at p_2 causes the value of y to drop to 3, at which point the monitor detects a property violation.

6. The DIANA Tool

We have implemented the above technique as a tool, called DIANA (Distributed ANALysis) (see Figure 1). DIANA is publicly available and can be downloaded from: <http://fsl.cs.uiuc.edu/diana/>. Both DIANA and the framework under which it operates are written in Java.

6.1. Actors

A number of formalisms can be used to reason about distributed systems, the most natural one being Actors [2, 3]. Actors are a model of distributed reactive objects and have a built-in notion of encapsulation and interaction, making them well suited to represent evolution and coordination between interacting components in distributed applications. Conceptually, an actor encapsulates a state, a thread of control, and a set of procedures which manipulate the state. Actors coordinate by asynchronously sending messages to each other. In the actor framework, a distributed system consists of different actors communicating through messages. Thus, there is an actor for each process in the system.

In the implementation, each type of actor (or process) is denoted by a Java class that extends a base class `Actor`. This base class implements a message queue and provides the method `send` for asynchronous message sending. Each actor object executes in a separate thread. The state of an actor is represented by the fields of the Java class. Each Java class also contains a set of public methods that can be invoked in response to messages received from other actors. A system level actor called `ActorManager` takes a message and transfers it to the message queue of the target actor. The target actor takes an available message from the message queue and invokes the method mentioned in the message.

While processing a message, an actor may send messages to other actors. Message sending, being asynchronous, never blocks an actor. However, an actor blocks if there is no message in its message queue. The system is initialized by the *ActorManager* object that creates all the actors in the system and starts the execution of the system.

6.2. Distributed Monitors in DIANA

The user of DIANA specifies the local PT-DTL formulae to be monitored on each actor in a special file. Each actor has a unique name, which is the name of the corresponding process. The name is passed as a string at the time creation of an actor.

As Figure 1 shows, the core of DIANA consists of two modules: an *instrumentation* module and a *monitoring* module. The instrumentation module takes the specification file and the distributed program written in the above framework and creates a Java class *MonitorImpl* that implements a local monitor for each actor (or process). It also automatically instruments the distributed program *at the bytecode level* (after compilation), so that the distributed program invokes its local monitor whenever it modifies a field variable (internal event), sends a message, or invokes a method (receive event).

One may alternately choose to evaluate the epistemic expressions and formulae immediately before sending an event (Subsection 5.1). In this case the local monitor is not invoked when field variables are modified. While runtime overhead was not the major concern for us in implementing our prototype, delaying such evaluation generally reduces the runtime overhead.

6.3. Test Cases

We implemented the following voting algorithm: a *Chair* process asks for vote on a resolution from N voters named *Voter₁*, *Voter₂*, ..., *Voter_N*, where N is initialized to an arbitrary but fixed positive number. We assume that the processes are connected in a tree kind of network with the *Chair* at the root of the tree and the voters at different nodes. Each voter randomly decides if it wants to vote for or against the resolution, and correspondingly stores 1 or 0 in a local state variable called *vote*. The voter then sends its decision to its immediate parent in the tree. The parent collects the votes and sends the sum of its vote and its progenies' votes to its immediate parent. The *Chair* process collects all the votes and rejects the resolution only if half or more voters have rejected. We monitor the following safety property at *Chair*:

$$\text{reject} \rightarrow ((\sum_{i \in [1..N]} @_{\text{Voter}_i}(\text{vote})) < N/2)$$

The property was found to be violated in several runs: at some voter nodes, the voter sent the sum of its progenies' votes without adding its own vote. This resulted in the rejection of the resolution when it should have been accepted.

We have also tested a vector clock [8, 15] algorithm implemented in the framework presented in this section. The algorithm was implemented as part of global snapshot and garbage collection algorithm. In this algorithm, each process is assumed to have a local vector clock V that it updates according to the standard vector clock algorithm [8] whenever there is an internal event, a send event or a receive event. The safety property that the algorithm must satisfy is that, at every process p_i : “all entries of the local vector clock must be greater than or equal to the local vector clock in a causally latest preceding state of any other process,” expressed as the following *i*-formula:

$$\Box (\bigwedge_{j \in [1..n]} V \geq @_j V)$$

where $V \geq V'$ when every entry in V is greater than or equal to the corresponding entry in V' . Another safety property states that “at every process p_i the i -th entry in its local vector clock must be strictly greater than the i -th entry of the local vector clock of any other process”. This can be expressed as the following *i*-formula:

$$\Box (\bigwedge_{j \in [1..n]} V[i] > @_j V[i])$$

The second property was found to be violated in some computations due to a bug caused by failure to increment the i -th entry of the local vector clock of process p_i when receiving events.

These simple examples illustrate the practical utility and power of PT-DTL and the monitoring tool DIANA based on it.

7. Conclusion and Future Work

This work represents the first step in effective distributed monitoring. The work presented here suggests a number of problems that require further research. The logic itself could be made more expressive so that it expresses not only safety, but also liveness properties. One difficulty is that software developers are reluctant to use formal notations. A partial solution may be to merge the present work with a more expressive and programmer friendly monitoring temporal logic such as EAGLE [6]. A complementary approach is to develop visual notations and synthesizing temporal logic formulas from such notations. There may also be the possibility of learning formulas based on representative scenarios.

An interesting avenue of future investigation that our work suggests is what we call *Knowledge-based*

Aspect-Oriented Programming. Knowledge-based Aspect-Oriented Programming is a meta-programming discipline that is suitable for distributed applications. In this programming paradigm, appropriate actions are associated with each safety formula; these actions are taken whenever the formula is violated to guide the program and avoid catastrophic failures.

Acknowledgements

The first three authors are supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract F30602-00-2-0586, the DARPA IXO NEST Program, contract F33615-01-C-1907), the ONR Grant N00014-02-1-0715, and the Motorola Grant MOTOROLA RPS #23 ANT. The last author is supported in part by the joint NSF/NASA grant CCR-0234524.

References

- [1] *1st, 2nd and 3rd CAV Workshops on Runtime Verification (RV'01 - RV'03)*, volume 55(2), 70(4), 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science: 2001, 2002, 2003.
- [2] G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.
- [3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
- [4] R. Alur, D. Peled, and W. Penczek. Model checking of causality properties. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 90–100, San Diego, California, 1995.
- [5] R. Aumann. Agreeing to disagree. *Annals of Statistics*, 4(6):1236–1239, 1976.
- [6] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57, Venice, Italy, January 2004. Springer-Verlag.
- [7] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.
- [8] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (WPDD'88)*, pages 183–194. ACM, 1988.
- [9] K. Havelund and G. Roşu. Java pathexplorer – A runtime verification tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, Montreal, Canada, June 18 - 21, 2001.
- [10] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, 2002.
- [11] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: a run-time assurance tool for java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
- [12] M. Leucker. Logics for mazurkiewicz traces. Technical Report AIB-2002-10, RWTH, Aachen, Germany, April 2002.
- [13] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.
- [14] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
- [15] F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science, 1989.
- [16] B. Meenakshi and R. Ramanujam. Reasoning about message passing in finite state environments. In *International Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853 of *Lecture Notes in Computer Science*, pages 487–498. Springer-Verlag, 2000.
- [17] W. Penczek. A temporal approach to causal knowledge. *Logic Journal of the IGPL*, 8(1):87–99, 2000.
- [18] W. Penczek and S. Ambroszkiewicz. Model checking of causal knowledge formulas. In *Workshop on Distributed Systems (WDS'99)*, volume 28 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1999.
- [19] R. Ramanujam. Local knowledge assertions in a changing world. In *Theoretical Aspects of Rationality and Knowledge (TARK'96)*, pages 1–14. Morgan Kaufmann, 1996.
- [20] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC'03)*, Helsinki, Finland, 2003.
- [21] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, September 2000.
- [22] P. S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for Mazurkiewicz traces. In *Twelfth Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, pages 183–194, Warsaw, Poland, 1997.

On Efficient Communication and Service Agent Discovery in Multi-agent Systems

Myeong-Wuk Jang and Gul Agha

Department of Computer Science

University of Illinois at Urbana-Champaign

{mjang, agha}@uiuc.edu

Abstract

The paper studies two closely related problems: how to support efficient message passing in large-scale agent systems given that agents are mobile, and how to facilitate the discovery of service agents in an open environment where agents may enter and leave. The Actor Architecture has been designed to study simulations of large-scale agent systems where agents obey the operational semantics of actors. We describe the solutions to these two problems that have been adopted in the Actor Architecture. The problem of efficient message-passing is partially addressed by using dynamic names for agents. Specifically, a part of the name of a mobile agent changes continuously as a function of the agent platform that it is currently hosted by. This enables the agent platform of a sender to use location information about the receiver agent in order to optimize message delivery. The problem of agent discovery is addressed by using a broker agent. Moreover, the sender agent may reduce the communication that is required between the sender itself and a broker agent by sending the broker an agent to localize the search for the appropriate service agents. In order to mitigate security problems, this search agent is very restricted in what operations it is allowed to perform and is transmitted in the form of a passive object. A description of the Actor Architecture is given, focusing on these two ideas and their preliminary evaluation.

Keywords: Open Distributed System, Multi-agent System, Actor System, Message Passing, Brokering Service.

1. Introduction

A number of multi-agent systems, including EMAF [3], JADE [4], InfoSleuth [5], and OAA [6], support an *open agent systems*, i.e., systems in which agents may enter and leave at any time. Moreover, the growth of computational power and networks has made large-scale open agent systems a promising technology. However, before this vision of scalable open agent

systems can be realized, two closely related problems must be addressed:

- How can an agent efficiently discover service agents which are previously unknown? In an open agent system, the mail addresses or names of all agents are not globally known; agents may not have the addresses of other agents with whom they need to communicate. This suggests that *middle agent services* such as *brokering* and *matchmaking* are necessary [14]. As we scale up agent systems, efficiently implementing these services is a challenge.
- How to efficiently send messages to agents which have potentially moved? In mobile agent systems, efficiently sending messages to an agent is not simple because they move continuously from one agent platform to another. For example, one obvious solution, viz. requiring the *agent platform* on which a mobile agent is created to manage location information about that agent, may double the message passing overhead.

We address the message passing problem for mobile agents in part by providing a richer structure on names which allows the names to dynamically evolve. Specifically, the names of agents include information about their current location. Moreover, rather than simply sending data as messages, we allow an agent system to use the data to find the location of an appropriate receiver agent.

We have implemented our ideas in a Java-based agent system called the *Actor Architecture* (or AA). AA supports the *actor semantics* for agents: each agent is an autonomous process with a unique name (address), message passing between agents is asynchronous, new agents may be dynamically created, and agent names may be communicated [1]. AA is being used to develop tools to facilitate large-scale simulations, but it may be used for other large-scale open agent applications as well; AA has been designed with a modular and extensible, application-independent structure. The primary features of AA are to provide a light-weight implementation of agents, minimize communication

overhead between agents, and enable service agents to be located efficiently.

This paper is organized as follows. Section 2 introduces the overall structure and functions of AA and the agent life cycle model in AA. Section 3 explains how to reduce the communication overhead in AA, and Section 4 shows how to improve the middle agent service in AA. Section 5 describes our experiments with AA and evaluation of our approaches. Finally, in Section 6 we discuss our preliminary conclusions and research directions.

2. The Actor Architecture

AA provides a light-weight implementation of agents as active objects or actors [1]. Actors can provide the infrastructure for a variety of agent systems; they are social and reactive, but they are *not* explicitly required to be “autonomous” in the sense of being proactive [16]. However, autonomous actors may be implemented in AA and many of our experimental studies require proactive actors. Although the term agent has been used to mean proactive actors, for our purposes, the distinction is not critical. In this paper, we use terms ‘agents’ and ‘actors’ as synonyms.

The Actor Architecture consists of two main components:

1. Actor execution environments called *AA platforms*. AA platforms provide the system environment in which actors exist and interact with other actors. Specifically, AA platforms provide actor state management, actor communication, actor migration, and middle actor services.
2. An *actor library* which supports the development of agents that are executed on AA platforms.

We describe the structure of AA in greater detail. An AA platform consists of eight components (see Figure 1): Message Manager, Transport Manager, Transport Sender, Transport Receiver, Delayed Message Manager, Actor Manager, Actor Migration Manager, and ATSpace.

A *Message Manager* (MM) handles message passing between actors. Every message passes through at least one Message Manager. If the *receiver* actor of a message exists on the same AA platform, the MM of the platform directly delivers the message to the receiver actor. However, if the receiver actor is not on the same AA platform, this MM delivers the message to the MM of the platform where the receiver currently resides, and finally the MM delivers the message to the receiver. A *Transport Manager* (TM) maintains a public port for message passing between different AA platforms. When a *sender* actor sends a message to a receiver actor on a different AA platform, the *Transport Sender* (TS) residing on the same platform as the sender receives the message from the MM of the sender actor and delivers it to the *Transport Receiver* (TR) on the

AA platform of the receiver. When there is no a built-in connection between these two AA platforms, the TS contacts the TM of the AA platform of the receiver actor to open a connection so that the TM can create a TR for the new connection. Finally, the TR receives the message and delivers it to the MM on the same platform.

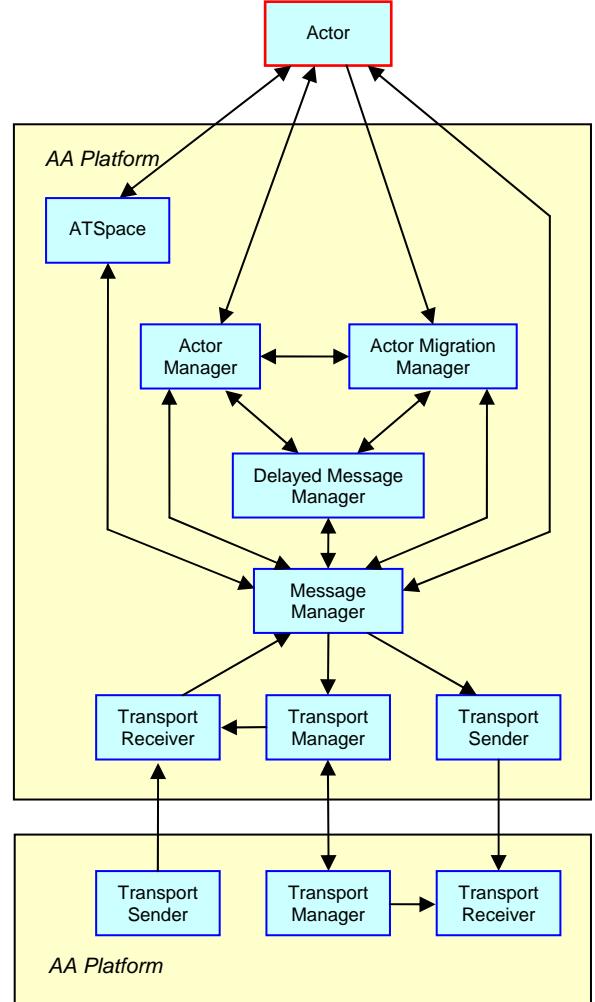


Figure1. The Architecture of an AA Platform

A *Delayed Message Manager* (DMM) temporarily holds messages for mobile actors while they are moving from their AA platform to other AA platforms. An *Actor Manager* (AM) manages states of the actors that are currently executing and the locations of the mobile actors created on the AA platform. An *Actor Migration Manager* (AMM) manages actor migration.

An *ATSpace* provides middle actor services, such as matchmaking and brokering services. Unlike other system components, an ATSpace is implemented as an actor. Therefore, any actor can create an ATSpace, and hence, an AA platform may have more than one ATSpaces. The ATSpace created by an AA platform is called the *default ATSpace* of the platform, and all actors can obtain the actor names of default ATSpaces. Once an actor has the name of an ATSpace, the actor

may send the ATSpace messages in order to use its services.

In AA, actors are implemented as active objects and executed as threads; actors on an AA platform are executed with that AA platform as part of one process. Each actor has one actor life cycle state at any time (see Figure 2). An actor may be *static*, meaning that it exists on its original AA platform, or it may be *mobile*, meaning that it has migrated from its original AA platform. The state information of a static actor appears within only its original AA platform while that of a mobile actor appears both on its original AA platform and on its current AA platform. When an actor is ready to process a message its state becomes *Active* and stays while the actor is processing the message. When a mobile actor initiates migration, its state is changed to *Transit*. Once the migration ends and the actor restarts, its state becomes *Active* on the current AA platform, and *Remote* on the original AA platform. Following a user request, an actor in the *Active* state may be *Suspended* state. In contrast to other agent life cycle models (e.g. [7, 12]), AA's life cycle model uses the *Remote* state to indicate that an agent that was created on the current AA platform is working on another AA platform.

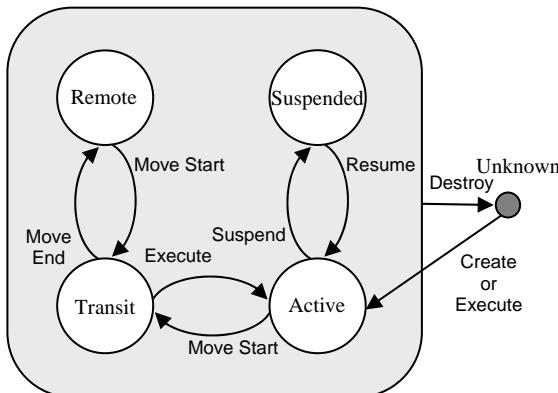


Figure 2. The Actor Life Cycle Model

3. Optimized Communication

We describe the mechanisms used to support actor communication. Specifically, AA uses two approaches to reduce the communication overhead for mobile actors that are not on their original AA platforms: namely, *location-based* message passing and *delayed* message passing.

3.1. Message Passing between Actors

After a message has been created, the message is managed by the Message Manager. When the receiver actor of a message is located on the same AA platform where the sender actor exists, the message is directly delivered to the receiver actor by the Message Manager (Figure 3a). However, if the receiver is on a different

machine, the message is delivered to the receiver through the Message Manager and the Transport Sender of the sender actor, and the Transport Receiver and the Message Manager of the receiver actor (Figure 3b). Although these two approaches of message passing are different at the system level, they are transparent to actors, and hence, actors always use the same operator to send their messages.

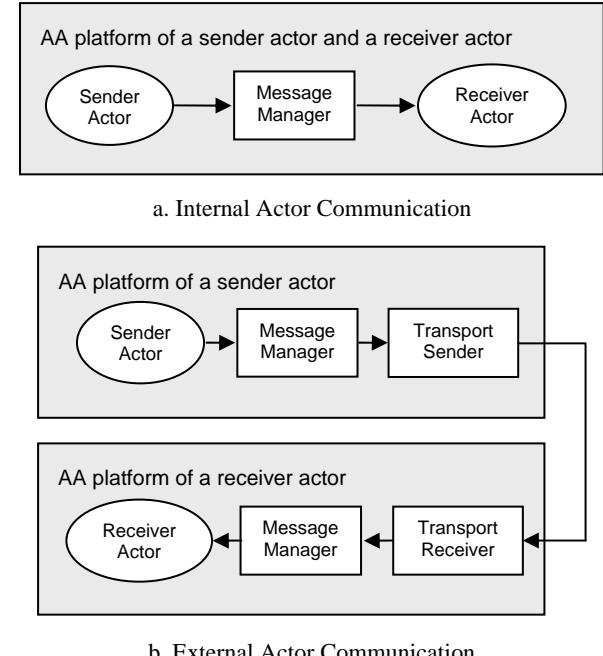


Figure 3. Procedure for Actor Communication

3.2. Location-based Message Passing

Before an actor can send messages to other actors, it must know the names of the intended receiver actors. In AA, each actor has its own unique name called *UAN* (*Universal Actor Name*). The UAN of an actor includes the *location information* and *unique identification number* of the actor as follows:

`uan://128.174.245.49:37`

From the above name, we can infer that the actor exists on the host whose IP address is 128.174.245.49, and that the actor is distinguished from other actors on the same platform with its unique identification number 37.

When the Message Manager of a sender actor receives a message whose receiver actor has the above name, it checks whether the receiver actor exists on the same AA platform. If they are on the same AA platform, the Message Manager finds the receiver actor on the AA platform and delivers the message. If they are not, the Message Manager of the sender actor delivers the message to the Message Manager of the receiver actor. In order to find the AA platform where the Message Manager of the receiver actor exists, the location information 128.174.245.49 in the UAN of the receiver actor is used. When the Message Manager on

the AA platform with IP address 128.174.245.49 receives the message, it finds the receiver actor and delivers the message.

The above actor naming and message delivery scheme works correctly when all actors are static. However, because an actor may migrate from one AA platform to another, we extend the basic behavior of the Message Manager with a *forwarding* service; when a Message Manager receives a message for a mobile actor, it delivers the message to the current AA platform of the mobile actor. To facilitate this service, an AA platform maintains a table providing the current locations of mobile actors that were created on the AA platform.

The problem with using only the universal names of actors for message delivery is that every message for a mobile actor that has moved from the original AA platform where the actor was created still has to pass through the original AA platform (Figure 4a). This kind of indirection may happen even in case the receiver actor exists on an AA platform that is close to (or the same as) the AA platform of the sender actor. In fact, message passing between actor platforms is relatively expensive. AA uses *Location-based Actor Naming* (LAN) for mobile actors in order to generally eliminate the need for this kind of indirection. Specifically, a LAN of an actor consists of its current location and its UAN as follows:

lan://128.174.244.147//128.174.245.49:37

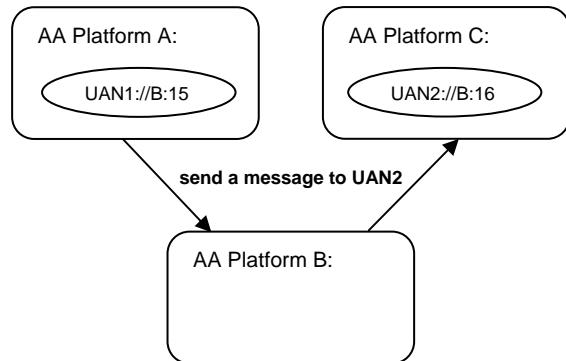
The current location of a mobile actor is set by an AA platform when the actor arrives on the AA platform. If the current location is the same as the location where an actor was created, the LAN of the actor does not have any special information beyond its UAN.

Under the location-based message passing scheme, when the Message Manager of a sender actor receives a message for a remote actor that exists on the different AA platform, it checks the current location of the receiver actor with its LAN and delivers the message to the AA platform where the receiver actor exists (Figure 4b). The rest of the procedure for the message passing is similar to that in UAN-based message passing scheme.

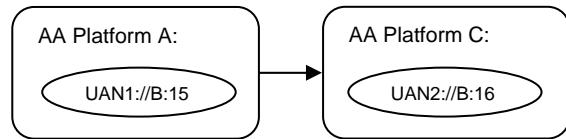
With location-based message passing, the system is more fault-tolerant; since messages for a mobile actor need not pass through the original AA platform of the actor, the messages may be correctly delivered to the actor even when the actor's original AA platform is not working correctly.

In order to use LAN address scheme, the location information in a LAN should be correct. However, mobile actors may move repeatedly, and a sender actor may have old LANs of mobile actors. Thus a message for a mobile actor may be delivered to the *previous AA platform* from where the actor left. This problem is addressed by having an old AA platform deliver the message to the original AA platform where the actor was created; the original platform always manages the

current address of an actor. There is no guarantee that the location-based message passing will perform better than the UAN-based message passing. Therefore, AA allows an actor to decide which addressing scheme is better for the current situation.



a. UAN-based Message Passing



b. Location-based Message Passing

Figure 4. Message Passing to a Mobile Actor

3.3. Delayed Message Passing

While a mobile actor is moving from one AA platform to another, the current AA platform of the actor is not well defined. Therefore, when the Message Manager of the original AA platform receives a message for a mobile actor, it sends the message to the Message Manager of the old AA platform. After the Message Manager of the old AA platform receives the message, it forwards the message to the Message Manager of the original AA platform because it no longer has information about the mobile actor's current location. An AA platform manages location information about only the mobile actors that are created on it. Thus, a message is continuously passed between these two AA platforms until the mobile actor updates location information with its new AA platform by informing the Actor Manager of the original AA platform.

In order to avoid unnecessary message passing, a Delayed Message Manager in AA platform is used; the Message Manager of the old AA platform delays the

message passing for a mobile actor while the state of the actor is *Transit*. For this operation, the Actor Manager of the old AA platform manages the state of the mobile actor and the Delayed Message Manager holds messages for the mobile actor until the actor reports that its migration has ended. After an actor finishes its migration, the new AA platform of the actor sends its old AA platform and its original AA platform a message to inform that the migration process has ended. Whenever one of these two AA platforms receives a message, the original AA platform changes the state of the mobile actor from *Transit* to *Remote* while the old AA platform removes information about the mobile actor.

4. Active Brokering Service

A brokering service is useful for supporting *attribute-based communication* between agents that are in an open multi-agent system. Recall that in open multi-agent systems, service agents that support a specific service may not be known to client agents; with attribute-based communication, client agents may use the attributes of the service they require instead of using the names of the service agents. The attributes of the service are delivered to a middle agent as a *tuple template*, and the middle agent tries to find a service agent or a set of service agents whose attributes are matched with the tuple template. The agents selected by the middle agent receive the message sent by the client agent through the middle agent. This service is very effective in open multi-agent systems, but the searching ability of the middle agent is often very restrictive for efficiency reasons; a middle agent typically provides only template-based exact matching or regular expression matching [2, 8, 11]. If a client agent requires a more powerful search, the client agent must use a matchmaking service instead of a brokering service; the client receives all the information about service agents and utilizes its own searching algorithm to locate proper service agents. For example, consider a middle agent that has information about seller agents with their products and prices, and a buyer agent wants to find seller agents that sell a computer with price greater than \$500 and less than \$1,000. If the exact matching service of the middle agent is not powerful enough to support this function, the buyer agent has to obtain all information about seller agents from the middle agent, and then choose seller agents that sell their computers within the price range. This sequence of operations requires moving of all information about seller agents from the middle agent to the buyer agent through the network.

In order to reduce the communication overhead, AA provides an active brokering service through an ATSpace actor. An ATSpace actor allows a sender actor to send its own search algorithm instead of a simple description for attributes of the service to locate receiver actors, and the algorithms are executed in the

ATSpace actor. In Figure 5, the seller actors with UAN2 and UAN3 are selected by the search algorithm, and the ATSpace actor delivers *sendComputerBrand* message to the actors. Finally, they will send information about brand names of their computers to the buyer actor.

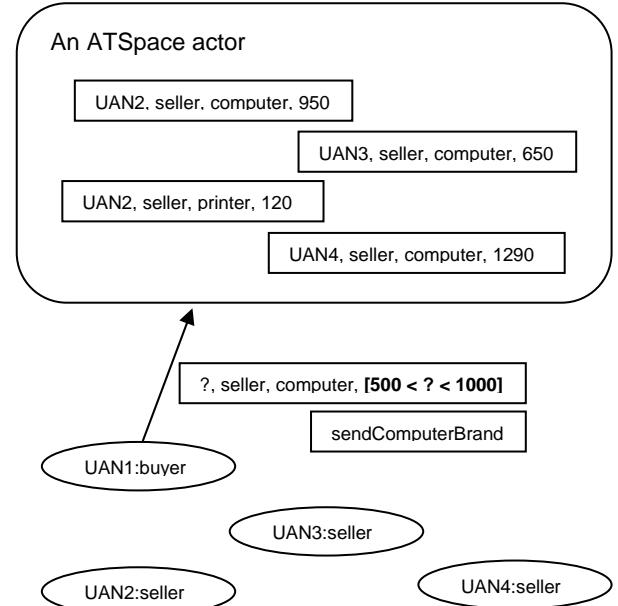


Figure 5. An Example of Active Brokering Service

In many situations, moving the search algorithm from the seller agent to the middle agent is less expensive than moving all the information about certain agents from the middle agent to the sender agent. Since a matching algorithm is provided by a sender agent and the algorithm is executed on a middle agent, the middle agent called an *ATSpace* actor can provide application oriented brokering service more efficiently. Moving the search algorithm may be accomplished by sending an agent incorporating the search algorithm. However, this extension introduces security threats for the data in the ATSpace actor. AA provides some solutions to mitigate such threats, in particular by not allowing arbitrary agents to be sent; agents are sent as passive objects and their functions are controlled by the ATSpace actor [9].

5. Experiments and Evaluation

AA platforms and actors have been implemented in Java language to support operating system independent actor mobility. We are using our actor system for large-scale UAV (Unmanned Aerial Vehicle) simulations. In these simulations, we investigate the effects of different collaboration behaviors among the large number of micro UAVs during their surveillance missions on the large number of targets [10]. For our experiments, we execute more than 5,000 actors on four computers: 2500 micro UAVs, 2500 targets, and other simulation purpose actors.

The delayed message passing removes unnecessary message passing for moving agents. When the delayed message passing is used, the old AA platform of a mobile actor needs to manage its state information until the actor finishes its migration, and the new platform of the mobile actor needs to report the migration state of the actor to its old AA platforms. In our experience, this overhead is more than compensated; without the delayed message passing the same message may get delivered seven or eight times between the original AA platform and the old AA platform in the local network environment while a mobile actor is moving. If a mobile actor takes more time for its migration, this number may be even greater. Moreover, the extra hops also make the message log files more complex and reduce their readability.

The performance benefit of the active brokering service can be measured by comparing it with the matchmaking service that provides the same service along four different dimensions: the number of messages, the total size of messages, the total size of memory space on two AA platforms for client and middle actors, and the time for the whole operation. First, in the matchmaking service, the number of messages is $n + 2$, where n is the number of service actors, while it is $n + 1$ in the active brokering service. In the former, the number of messages for this operation includes a service request message from the client actor to the middle actor, a reply message from the middle actor to the client actor, and multicast messages from the client actor to n service actors. The active brokering service does not require the reply message, and hence, one message is unnecessary. It is a small difference, but more significantly, the total size of messages is very different. The service request message in the active brokering service is a little larger than that in the matchmaking service, because it includes the code for a searching algorithm and the message to be delivered to service actors. However, the reply message in the matchmaking service to be communicated across the network may be much larger than the difference of service request messages in two approaches. Moreover, the total size of storage space for the active brokering service is less than that in the matchmaking service; in the matchmaking service case a copy of the data exists in the client actor, while in the active brokering service such a copy need not exist in the client actor. However, for the data safety, the active brokering service may still keep a copy of the data. Finally, the difference in operation times except communication times is relatively small. Mainly, the computation in matchmaking is off-loaded to the server side. However, since the communication time is proportioned to the total size of messages, the active brokering service is more efficient in the time for the whole operation.

6. Conclusions

The location-based message passing scheme in AA reduces the number of hops (AA platforms) that a message for a mobile actor goes through. The basic mechanism of the location-based message passing is similar to the message passing in Mobile IP [13], although its application domain is different from ours. The original and current AA platforms of a mobile actor correspond to the home and foreign agents of a mobile client in Mobile IP, and the UAN and LAN of a mobile actor are similar to the home address and care-of address of a mobile client in Mobile IP. However, while the sender node in Mobile IP manages a binding cache to map home addresses to care-of addresses, the sender AA platform in AA does not have a mapping table, and while the home agent communicates with the sender node to update the binding cache, it does not happen in AA.

Our work may also be compared to SALSA. In SALSA, a sender actor may use a middle actor called Universal Actor Naming Server to locate the receiver actor [15]. SALSA's approach requires the receiver actor to register its location at a certain middle actor, and the middle actor must manage the mapping table. With the location-based message passing scheme in AA, a LAN of an actor is changed automatically as a function of an AA platform, and the mapping table does not exist at any single place.

We are currently implementing and testing new message passing mechanisms for mobile agents. For example, the location-based message passing may be modified to allow a mobile agent to set its future location address in its LAN and announce this to other agents. For the delayed message passing, instead of the old AA platform of a mobile agent, the new AA platform of the mobile agent may hold messages for the agent, and hence, when the agent finishes its migration it receives the messages managed by the Delayed Message Manager of the AA platform. We plan to investigate various trade-offs and methods for automatically selected best estimated message-passing mechanism for a given situation.

Acknowledgements

This research is sponsored by the Defense Advanced Research Projects Agency under contract number F30602-00-2-0586.

References

- [1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass, 1986.
- [2] G. Agha and C.J. Callsen, "ActorSpaces: An Open Distributed Programming Paradigm," *Proceedings of the 4th*

ACM Symposium on Principles & Practice of Parallel Programming, May 1993, pp. 23-32.

[3] S. Baeg, S. Park, J. Choi, M. Jang, and Y. Lim, "Cooperation in Multiagent Systems," *Intelligent Computer Communications (ICC '95)*, Cluj-Napoca, Romania, June 1995, pp. 1-12.

[4] F. Bellifemine, A. Poggi, and G. Rimassa, "JADE - A FIPA-compliant Agent Framework," *Proceedings of Practical Application of Intelligent Agents and Multi-Agents (PAAM '99)*, London, April 1999, pp. 97-108.

[5] R.J. Bayardo Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk, "InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments," *ACM SIGMOD Record*, Vol. 26, No. 2, June 1997, pp. 195-206.

[6] P.R. Cohen, A.J. Cheyer, M. Wang, and S. Baeg, "An Open Agent Architecture," *AAAI Spring Symposium*, March 1994, pp. 1-8.

[7] Foundation for Intelligent Physical Agents, *SC00023J: FIPA Agent Management Specification*, December 2002.
<http://www.fipa.org/specs/fipa00023/>

[8] N. Jacobs and R. Shea, "The Role of Java in InfoSleuth: Agent-based Exploitation of Heterogeneous Information Resources," *Proceedings of Intranet-96 Java Developers Conference*, April 1996.

[9] M. Jang, A. Abdel Momen, and G. Agha, "ATSpace: A Middle Agent to Support Application-Oriented Matchmaking and Brokering Services," Technical Report UIUCDCS-R-

2004-2430, Department of Computer Science, University of Illinois at Urbana-Champaign, April 2004.

[10] M. Jang, S. Reddy, P. Totic, L. Chen, and G. Agha, "An Actor-based Simulation for Studying UAV Coordination," *15th European Simulation Symposium (ESS 2003)*, October 2003, pp. 593-601.

[11] D.L. Martin, H. Oohama, D. Moran, and A. Cheyer, "Information Brokering in an Agent Architecture," *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, April 1997, pp. 467-489.

[12] D.G.A. Mobach, B.J. Overeinder, N.J.E. Wijngaards, and F.M.T. Brazier, "Managing Agent Life Cycles in Open Distributed Systems," *Proceedings of the 2003 ACM symposium on Applied computing*, Melbourne, Florida, 2003, pp. 61-65.

[13] C.E. Perkins, "Mobile IP," *IEEE Communications Magazine*, Vol. 35, No. 5, May 1997, pp. 84-99.

[14] K. Sycara, K. Decker, and M. Williamson, "Middle-Agents for the Internet," *Proceedings of the 15th Joint Conference on Artificial Intelligences (IJCAI-97)*, 1997, pp. 578-583.

[15] C.A. Varela and G. Agha, "Programming Dynamically Reconfigurable Open Systems with SALSA," *ACM SIGPLAN Notices: OOPSLA 2001 Intriguing Technology Track*, Vol. 36, No. 12, December 2001, pp. 20-34.

[16] M. Wooldridge, *An Introduction to MultiAgent Systems*, John Wiley & Sons, Ltd, 2002.

On Specifying and Monitoring Epistemic Properties of Distributed Systems

Koushik Sen, Abhay Vardhan, Gul Agha, Grigore Roşu

Department of Computer Science

University of Illinois at Urbana Champaign

{ksen, vardhan, agha, grosu}@cs.uiuc.edu

Abstract

We present an epistemic temporal logic which is suitable for expressing safety requirements of distributed systems and whose formulae can be efficiently monitored at runtime. The monitoring algorithm, whose underlying mechanism is based on symbolic knowledge vectors, is distributed, decentralized and does not require any messages to be sent solely for monitoring purposes. These important features of our approach make it practical and feasible even in the context of large scale open distributed systems.

1. Introduction

The discovery and prevention of software errors is a difficult problem involving many different aspects, such as incorrect or incomplete specifications, errors in coding, faults and failures in the hardware, operating system or network. Two prominent formal approaches used in checking for errors are: theorem proving and model checking. Theorem proving is powerful but labor-intensive, requiring intervention by someone with fairly sophisticated mathematical training. On the other hand, model checking is more of a push-button technology, but despite exciting recent advances, the size of systems for which it is feasible remains rather limited. As a result, most system builders continue to rely on testing to identify bugs in their implementation.

There are two problems with software testing. First, testing is generally done in an *ad hoc* manner: it requires the software developer to translate properties into specific checks on the program state. Second, test coverage is rather limited. To mitigate the first problem, software often includes dynamic checks on the systems state to identify problems at run-time. Recently, there has been some interest in run-time monitoring techniques [1] which provide a little more rigor in testing. In this approach, monitors are automatically synthesized from a formal specification. These monitors may then be deployed off-line for debugging or on-line for dynamically checking that safety properties are not being violated during system execution.

In [6] we argue that distributed systems may be effectively monitored against formally specified safety requirements. By effective monitoring we mean not only linear efficiency, but also decentralized monitoring where few or no

additional messages need to be passed for monitoring purposes. We introduced an epistemic temporal logic for distributed knowledge, called *past time linear temporal logic* and abbreviated PT-DTL, and showed how monitors can be synthesized for it. PT-DTL formulae are local to particular processes and are interpreted over projections of global state traces that the current process *is aware of*. In this paper, we increase the expressiveness of PT-DTL and make it more programmer friendly by adding constructs similar to value binding in programming languages and quantification in first order logic. These constructs allow us to succinctly specify properties of open distributed systems involving data. The new logic is called xDTL and its novel features are inspired from EAGLE [3].

Let us assume an environment in which a node a may send a message to a node b requesting a certain value. The node b , on receiving the request, computes the value and sends it back to a . There can be many such nodes, any pair can be involved in such a transaction, but suppose that a crucial property to enforce is that no node receives a reply from another node to which it had not issued a request earlier. One can check this global property by having one local monitor on each node, which monitors a single property. For instance, a monitors “if a has received a value from b then it must be the case that previously in the past at b the following held: b has computed the value and at a a request was made for that value in the past”. Using xDTL, all one needs to do is to provide the safety policy as a formula:

```
valueReceived →
@b(◊(valueComputed ∧ @a(◊valueRequested)))
```

$@$ is an *epistemic operator* and should be read “at”; $@_b F$ is a *remote property* that should be thought of as the value of F in the most recent local state of b that the current process is aware of. In PT-DTL[6], $@$ can only take one process as a subscript. In xDTL, as described later in the paper, $@$ can take any set of processes as a subscript together with a universal or an existential quantifier, so $@_b$ becomes “syntactic sugar” for $@_{\forall\{b\}}$ (or for $@_{\exists\{b\}}$). \Diamond should be read “eventually in the past”. Monitoring the formula above will involve sending no additional messages but only a few bits of infor-

mation piggybacked on the messages already being passed for the computation.

Suppose that we want to restrict the above safety policy by imposing a further condition that the value received by a must be same as the value computed by b . To express this stronger property, we need to compare values in states at two process that are not directly related. This property cannot be directly expressed in PT-DTL without introducing extra variables in the program itself. However, adding extra variables in the program can potentially result in side-effects which are not desirable. An elegant way to solve the problem is to introduce the notion data-binding in the logic used for monitoring. Informally, we can restate the property as follows: a monitors “if a has received a value from b then remember the value received in a variable k and it must be the case that previously in the past at b the following held: b has computed the value and the computed value is equal to k and at a a request was made for that value in the past”. This can be written formally as follows:

```
valueReceived → let  $k = \text{value}$  in
   $\text{@}_b(\Diamond(\text{computedValue} \wedge (k = \text{valueComputed}) \wedge \text{@}_a(\Diamond\text{requestedValue}))$ 
```

Informally, the construct “let $\vec{k} = \vec{\xi}$ in F ” binds the value of the expressions $\vec{\xi}$ at process a with the logic variables \vec{k} which can be referred by any expression in the formula F .

Another example in [6] regards monitoring certain correctness requirement in a leader-election algorithm. The key requirement for leader election is that there is at-most one leader. If there are 3 processes namely a, b, c and state is a variable in each process that can have values `leader`, `loser`, `candidate`, `sleep`, then we can write the property at every process as: “if a leader is elected then if the current process is a leader then, to its knowledge, none of the other processes is a leader”. We can formalize this requirement as the following PT-DTL formula at process a :

```
leaderElected → (state = leader →
   $\text{@}_b(\text{state} \neq \text{leader}) \wedge \text{@}_c(\text{state} \neq \text{leader})$ )
```

We can write similar formulae with respect to b and c . Given an implementation of the leader election problem, one can monitor each formula locally, at every process. If violated then clearly the leader election implementation is incorrect.

However, the above formula does not specify the requirement that every process must know the name of the process that has been elected as leader. We cannot express this stronger requirement in PT-DTL. However, using the construct “let $_$ in $_$ ” and assuming that the variable `leaderName` contains the name of the leader, the requirement can easily be stated in xDTL as follows:

```
leaderElected → let  $k = \text{leaderName}$  in
   $(\text{@}_b(\text{leaderName} = k) \wedge \text{@}_c(\text{leaderName} = k))$ 
```

Note that the above formula assumes that the name of every process involved in leader election is known to us beforehand. Moreover, the size of the formula depends on the number of processes. In a distributed system involving a large number of processes, writing such a large formula may be impractical. The problem becomes even more important in an open distributed system where we may not know the name of processes beforehand. To alleviate this difficulty, as already mentioned, we use a set of indices instead of a single index in the operator @_i . The set of indices denoting a set of processes can be represented compactly by a predicate on indices. For example, in the above formula, instead of referring to each process by its name we can refer to the set of all remote processes by the predicate $i \neq a$ and use this set as a subscript to the operator @_i :

```
leaderElected → let  $k = \text{leaderName}$  in
   $\text{@}_{\forall\{i|i \neq a\}}(\text{leaderName} = k)$ 
```

$\text{@}_{\forall\{i|i \neq a\}}(\text{leaderName} = k)$ denotes the fact that the formula $\text{leaderName} = k$ must hold true at all processes i satisfying the predicate $i \neq a$. This is equivalent to the first order logic formula $\forall i . ((i \neq a) \rightarrow \text{@}_i(\text{leaderName} = k))$.

The logic xDTL proposed in this paper, extending PT-DTL with the construct “let $_$ in $_$ ” and with quantified sets of processes in the subscript of the epistemic operator @_i , is more expressive and elegant than PT-DTL. These benefits are attained without sacrificing efficiency and the decentralized nature of monitoring.

Many researchers have proposed temporal logics to reason about distributed systems. Most of these logics are inspired by the classic work of Aumann [2] and Halpern *et al.* [4] on knowledge in distributed systems. Meenakshi *et al.* define a knowledge temporal logic interpreted over a message sequence charts in a distributed system [5] and develop methods for model checking formulae in this logic. However, in our work we address the problem of monitoring and investigate an expressive distributed temporal logic that can be monitored in a decentralized way.

The rest of the paper is organized as follows. Section 2 describes the basic concepts of distributed systems. Section 3 introduces the more expressive PT-DTL which we call xDTL. In Section 4 we conclude by briefly sketching a decentralized monitoring algorithm.

2. Distributed Systems

We consider a distributed system as a collection of processes, each having a unique name and a local state, communicating with each other through asynchronous message exchange. The computation of each process is abstracted out in terms of *events* which can be of three types: *internal*, an event denoting local state update of a process, *send*, an event denoting the sending of a message by a process to another process, and *receive*, an event denoting the reception

of a message by a process. Let E_i denote the set of events of process i and let E denote $\bigcup_i E_i$. Also, let $\lessdot \subseteq E \times E$ be defined as follows.

1. $e \lessdot e'$ if e and e' are events of the same process and e happens immediately before e' ,
2. $e \lessdot e'$ if e is the send event of a message at some process and e' is the corresponding receive event of the message at the recipient process.

The partial order \prec is the transitive closure of the relation \lessdot . This partial order captures the *causality* relation among the events in different processes and gives an abstraction of the *distributed computation* denoted by $\mathcal{C} = (E, \prec)$. In what follows, we assume an arbitrary but fixed distributed computation \mathcal{C} . Let us define \preceq as the reflexive and transitive closure of \lessdot . In Fig. 1, $e_{11} \lessdot e_{23}$ and therefore also $e_{11} \preceq e_{23}$. However, even though $e_{12} \not\prec e_{23}$, we have $e_{12} \preceq e_{23}$ as process 2 gets a message from process 3 which contains knowledge of e_{12} .

The *local state* of a process is abstracted out in terms of a set of events. For $e \in E$ we define $\downarrow e \stackrel{\text{def}}{=} \{e' \mid e' \preceq e\}$, that is, $\downarrow e$ is the set of events that causally precede e . For $e \in E_i$, we can think of $\downarrow e$ as the local state of process i when the event e has just occurred.

We extend the definition of \lessdot , \prec and \preceq to local states such that $\downarrow e \lessdot \downarrow e'$ iff $e \lessdot e'$, $\downarrow e \prec \downarrow e'$ iff $e \prec e'$, and $\downarrow e \preceq \downarrow e'$ iff $e \preceq e'$. We use the symbols s_i, s'_i, s''_i and so on to represent the local states of process i . We also assume that each local state s_i of each process i associates values to some local variables V_i , and that $s_i(v)$ denotes the value of a variable $v \in V_i$ in the local state s_i at process i .

We use the notation $causal_j(s_i)$ to refer to the latest state of process j of which process i knows while in state s_i . Formally, $causal_j(s_i) = s_j$ where s_j is a state at process j such that $s_j \preceq s_i$ and for all states s'_j in process j with $s'_j \preceq s_i$ we have $s'_j \preceq s_j$. For example, in Figure 1 $causal_1(\downarrow e_{23}) = \downarrow e_{12}$. Note that if $i = j$ then $causal_j(s_i) = s_i$.

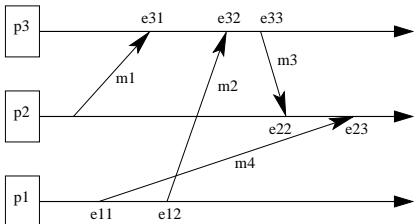


Figure 1. Sample Distributed Computation

3. Extended Distributed Temporal Logic

In order to reason about the global distributed computation locally, xDTL has a set of three new variants of *epistemic operators*, whose role is to evaluate an expression or a

formula in the *last known state* of a remote process. We call such an expression or a formula *remote*. In addition to the epistemic operators, we add the construct “let $\vec{k} = \vec{\xi}$ in F ” to xDTL to bind expressions to local logic variables that can be referred by any expression or formula in F .

The intuition underlying xDTL is that each process may be associated a local formula which, due to the epistemic operators, can refer to the global state of the distributed system. These formulae are required to be valid at the respective processes during a distributed computation. The distributed computation satisfies the specification when all the local formulae are shown to satisfy the computation. Next, we formally describe the syntax and semantics of xDTL.

3.1. Syntax

In the sequel, whenever we talk about an xDTL formula, it is in the context of a particular process, having the name i . We call such formulae *i-formulae* and let F_i, F'_i , etc., denote them. Additionally, we introduce the notion of expressions local to a process i called as *i-expressions* and let ξ_i, ξ'_i , etc., denote them. Informally, an *i-expression* is an expression over the global state of the system that process i is currently aware of. Local predicates on *i-expressions* form the atomic propositions on which the temporal *i-formulae* are built.

We add the *epistemic operators* $@_{\forall J} F_j$ and $@_{\exists J} F_j$ which is true if at all (or some, respectively) processes j in the set J , F_j holds. Similarly, we add the epistemic operator $@_{J\xi_j}$ which returns the set of j -expressions ξ_j for all processes j in the set J . The sets J can be expressed compactly using predicates over j . For example, J can be the sets $\{j \mid j \neq a\}$ or $\{j \mid \text{client}(j)\}$. The following gives the formal syntax of xDTL with respect to a process i , where i and j are the name of any process (not necessarily distinct):

$F_i ::=$	$\text{true} \mid \text{false} \mid P(\vec{\xi}_i) \mid \neg F_i \mid F_i \text{ op } F_i$	propositional
	$\mid \odot F_i \mid \Diamond F_i \mid \Box F_i \mid F_i \mathcal{S} F_i$	temporal
	$\mid @_{\forall J} F_j \mid @_{\exists J} F_j$	epistemic
	$\mid \text{let } \vec{k} = \vec{\xi}_i \text{ in } F_i$	binding
$\xi_i ::=$	$c \mid v_i \mid k \mid f(\vec{\xi}_i)$	functional
	$\mid @_{J\xi_j}$	epistemic
$\vec{\xi}_i ::=$	(ξ_i, \dots, ξ_i)	

The infix operator op can be any binary propositional operator such as $\wedge, \vee, \rightarrow, \equiv$. The term $\vec{\xi}_i$ stands for a tuple of expressions on process i . The term $P(\vec{\xi}_i)$ is a (computable) predicate over the tuple $\vec{\xi}_i$ and $f(\vec{\xi}_i)$ is a (computable) function over the tuple. For example, P can be $<, \leq, >, \geq, =$. Similarly, some examples of f are $+, -, /, *$. Variables v_i belong to the set V_i containing all the local state variables of process i . c stays for constants, e.g., 0, 1, 3.14.

3.2. Semantics

The semantics of xDTL extends the semantics of PT-DTL by defining the three variants of epistemic operators

$\mathcal{C}, s_i, [e] \models @_{\forall J} F_j$	iff $\forall j . (j \in J) \rightarrow \mathcal{C}, s_j, [e] \models F_j$ where $s_j = causal_j(s_i)$
$\mathcal{C}, s_i, [e] \models @_{\exists J} F_j$	iff $\exists j . (j \in J) \wedge \mathcal{C}, s_j, [e] \models F_j$ where $s_j = causal_j(s_i)$
$\mathcal{C}, s_i, [e] \models \text{let } (k, \dots, k') = (\xi_i, \dots, \xi'_i) \text{ in } F_i$	iff $\mathcal{C}, s_i, [e, k \mapsto (\mathcal{C}, s_i, [e])[\xi_i]], \dots, k' \mapsto (\mathcal{C}, s_i, [e])[\xi'_i]] \models F_i$
$(\mathcal{C}, s_i, [e, k \mapsto val])[\![k]\!]$	$= val$
$(\mathcal{C}, s_i, [e])[\![@_J \xi_j]\!]$	$= \{(\mathcal{C}, s_j, [e])[\xi_j] \mid s_j = causal_j(s_i) \wedge j \in J\}$

Table 1. Semantics of xDTL

and the binding operator. The semantics is given by recursively defining the satisfaction relation $\mathcal{C}, s_i, [e] \models F_i$, where $[e]$ is an environment carrying the bindings for different logic variables which gets introduced by the “let _ in _” operator. $(\mathcal{C}, s_i, [e])[\xi_i]$ is the value of the expression ξ_i in the state s_i under the environment $[e]$. Table 1 formally gives the semantics of the new operators of xDTL. For the semantics of other operators the readers are referred to [6]. We assume that expressions are properly typed. Typically these types would be `integer`, `real`, `strings`, etc. We also assume that s_i, s'_i, s''_i, \dots are states of process i and s_j, s'_j, s''_j, \dots are states of process j .

4. Monitoring Algorithm

To monitor xDTL formulae in a decentralized way, we synthesize *distributed monitors* as follows. For each process there is a separate monitor, called a *local monitor*, which checks the local xDTL formulae and can attach additional information to any outgoing message. This information can subsequently be extracted by the local monitor on the receiving side without changing the underlying semantics of the distributed program. The local monitor of each process i maintains a KNOWLEDGEVECTOR data-structure KV_i , storing for each process j in the system the status of all the safety policy sub-formulae and sub-expressions referring to j that i is aware of. The knowledge vector KV_i is appended to any message sent by i . When performing an internal computation step, the status of the local formulae and expressions is automatically updated in the local knowledge vector. When receiving a message from another process, the knowledge vector is updated if the received message contains more recent knowledge about any process in the system. To do this, a sequence number needs also to be maintained for each process in the knowledge vector. Unlike [6], the entries of KNOWLEDGEVECTOR are symbolic expressions instead of values. This is due to the fact that all the logic variables referred in an expression or a formulae may not be available at the time of evaluation of the expression or the formula. Therefore, the evaluation of a formula or an expression may be partial, containing the various logic variables. The logic variables in these formulae or expressions are replaced by actual values once they become available. A detailed discussion of the algorithm is beyond the scope of

this short paper. However, readers are referred to [6, 3] for some of the similar ideas.

5. Conclusion

We believe that the logic xDTL presented in this paper is a powerful underlying specification formalism for distributed systems. Specifications expressed as xDTL formulae can be effectively monitored, even in the context of large scale open distributed systems. However, it is worthwhile to investigate other extensions that increase its expressiveness without sacrificing the efficiency of monitoring.

Acknowledgements

The first three authors are supported in part by the DARPA IPTO TASK Program, contract F30602-00-2-0586, the DARPA IXO NEST Program, contract F33615-01-C-1907, the ONR Grant N00014-02-1-0715, and the Motorola Grant RPS #23 ANT. The last author is supported in part by the joint NSF/NASA grant CCR-0234524.

References

- [1] *1st, 2nd and 3rd CAV Workshops on Runtime Verification (RV’01 - RV’03)*, volume 55(2), 70(4), 89(2) of ENTCS. Elsevier Science: 2001, 2002.
- [2] R. Aumann. Agreeing to disagree. *Annals of Statistics*, 4(6), 1976.
- [3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI’04)*, volume 2937 of LNCS.
- [4] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.
- [5] B. Meenakshi and R. Ramanujam. Reasoning about message passing in finite state environments. In *International Colloquium on Automata, Languages and Programming (ICALP’00)*, volume 1853 of LNCS.
- [6] K. Sen, A. Vardhan, G. Agha, , and G. Roşu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of 26th International Conference on Software Engineering (ICSE’04) (To Appear)*.

Statistical Model Checking of Black-Box Probabilistic Systems

Koushik Sen, Mahesh Viswanathan, Gul Agha
Department of Computer Science,
University of Illinois at Urbana-Champaign.
`{ksen,vmahesh,agha}@uiuc.edu`

Abstract. We propose a new statistical approach to analyzing stochastic systems against specifications given in a sublogic of continuous stochastic logic (CSL). Unlike past numerical and statistical analysis methods, we assume that the system under investigation is an *unknown, deployed black-box* that can be passively observed to obtain sample traces, but cannot be controlled. Given a set of executions (obtained by Monte Carlo simulation) and a property, our algorithm checks, based on statistical hypothesis testing, whether the sample provides evidence to conclude the satisfaction or violation of a property, and computes a quantitative measure (*p*-value of the tests) of confidence in its answer; if the sample does not provide statistical evidence to conclude the satisfaction or violation of the property, the algorithm may respond with a “don’t know” answer. We implemented our algorithm in a Java-based prototype tool called VESTA, and experimented with the tool using case studies analyzed in [15]. Our empirical results show that our approach may, at least in some cases, be faster than previous analysis methods.

1 Introduction

Stochastic models and temporal logics such as continuous stochastic logic (CSL) [1, 3] are widely used to model practical systems and analyze their performance and reliability. There are two primary approaches to analyzing the stochastic behavior of such systems: *numerical* and *statistical*. In the numerical approach, the formal model of the system is *model checked* for correctness with respect to the specification using symbolic and numerical methods. Model checkers for different classes of stochastic processes and specification logics have been developed [8, 14, 13, 4, 5, 2, 6]. Although the numerical approach is highly accurate, it suffers from being computation intensive. An alternate method, proposed by Younes and Simmons [16], is based on Monte Carlo simulation and sequential hypothesis testing. Being statistical in nature, this approach is less accurate and only provides probabilistic guarantees of correctness. The approach does not assume knowledge of a specific formal model for the system being analyzed, and therefore can be potentially applied to analyzing complex dynamical systems such as generalized semi-Markov processes (GSMPs), for which symbolic and numerical methods are impractical. However, the Younes and Simmons’ approach assumes that the system is controllable (not black-box) and can be used to generate sample executions from any state on need basis.

Both the numerical and the current statistical methods suffer from several serious drawbacks when it comes to analyzing practical systems. First, modern day systems are large heterogeneous, and assembled by integrating equipment

and software from diverse vendors, making the construction of a formal model of the entire system often impossible and thus limiting the feasibility of numerical and symbolic methods. Second, for large network systems, meaningful experiments may involve dozens or even thousands of routers and hosts, which would mean that the system needs to be deployed before reasonable performance measures can be obtained. However, once they are deployed, such systems cannot be controlled to generate traces from any state, making it impossible to generate execution samples on a need basis as is required by the Younes *et. al*'s statistical approach.

Despite the success of current analysis methods [10, 13, 12, 15, 8], there is therefore a need to develop methods to analyze stochastic processes that can be applied to deployed, unknown “black-box” systems (systems from which traces cannot be generated from any state on need)¹. In this paper we address these concerns by proposing a new statistical approach to model checking. Like in Younes *et. al*'s approach, discrete event simulation methods are used to obtain a set of sample executions; however, unlike their method we assume no control over the set of samples we obtain. We then test these samples using various statistical tests determined by the property to be verified. Since we assume that the samples are generated before testing, our algorithm relies on statistical hypothesis testing, rather than sequential hypothesis testing. Our inability to generate samples of our choosing and at the time of our choosing ensures that our approach differs from the previous statistical approach in one significant way: unlike the previous approach where the model checker’s answer can be guaranteed to be correct within the required error bounds, we instead compute a quantitative measure of confidence (the *p*-value, in statistical testing terminology [11]) in the model checker’s answer. Our algorithm computes the satisfaction of the desired formula by recursively determining the satisfaction of its subformulas (and the confidence of such an answer) in the “states” present in the sample. This presents a technical challenge because our algorithm, being statistical in nature, may be uncertain about the satisfaction of some formulas based on the given samples. The algorithm needs to compute useful answers (as far as possible) even in the presence of uncertain answers about the satisfaction of subformulas. We overcome this challenge by interpreting such “don’t know” answers in an adversarial fashion. Our algorithm, thus checks if the sample provides evidence for the satisfaction or violation of a property and the confidence with which such an assertion holds, or gives up and says “don’t know.” The algorithm that we propose suffers from one drawback when compared with the previous statistical approach. Since we analyze a fixed sample, we will get useful answers only when there are sufficient samples for each “relevant state.” Therefore our method is likely to work well only when a finite set of samples is enough to provide sufficient

¹ We assume that the samples generated from the system by discrete event simulation have information about the “system state”. We, however, make no assumptions about the transition structure of the underlying system, nor do we assume knowledge about the transition probabilities; the system under investigation is black-box in this sense.

information about relevant states. Examples of systems we can successfully analyze are Continuous-time Markov Chains (CTMCs) or systems whose relevant states are discrete, while we are unlikely to succeed for GSMPs in general.

A closely related approach to analyzing stochastic systems based on Monte Carlo simulation is by Herault et. al. [7], which can model-check discrete-time Markov chains against properties expressed in an expressively weak logic (“positive LTL”).

We have implemented the whole procedure in Java as a prototype tool, called VESTA (Verification based on Statistical Analysis).² We have experimented with VESTA by applying it to some examples that have been previously analyzed in [15] and the results are encouraging. However, we suspect that VESTA would require a lot more space, because it stores the entire collection of samples it is analyzing. Even though space was not a problem for the examples we tried, we suspect that it may become an issue later.

The rest of the paper is organized as follows. Section 2 defines the class of systems we analyze and the logic we use. In Section 3, we present our algorithm based on statistical hypothesis testing in detail. Details about VESTA and our case studies are presented in Section 4. In Section 5, we conclude and present possible directions for future research.

2 Preliminaries

2.1 Sample Execution Paths

The verification method presented here can be independent of the system model as long as we can generate sample execution paths of the system; the model and its definition are very similar to [16]. We will assume that the system being analyzed is some discrete event system that occupies some state $s \in S$, where S is the set of states of the system. The states in S that can effect the satisfaction of a property of our interest are called the “relevant states.” Note that the number of relevant states may be quite small compared to the whole state space of the system. For example, for a formula $\phi_1 \mathcal{U}^{\leq t} \phi_2$ the states that can be reached within time t are relevant. We assume that each relevant state can be uniquely identified and that information about a state’s identity is available in the executions. Since samples are generated before running our analysis algorithm, we require that a Monte Carlo simulation is likely to generate a sample that has enough “information” about the relevant states; if not our algorithm is likely to say that it cannot infer anything about the satisfaction of the property.

We assume that there is a labeling function L that assigns to each state a set of atomic propositions (from among those appearing in the property of interest) that hold in that state; thus $L : S \rightarrow 2^{AP}$, where AP is a set of relevant atomic propositions. The system remains in a state s until an event occurs, and then proceeds instantaneously to a state s' . An execution path that appears in our sample is thus a sequence

$$\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \dots$$

² Available from <http://osl.cs.uiuc.edu/~ksen/vesta/>

where s_0 is the unique initial state of the system, s_i is the state of the system after the i th event and t_i is the time spent in state s_i . If the k th state of this sequence is absorbing, then $s_i = s_k$ and $t_i = \infty$ for all $i \geq k$.

We denote the i th state in an execution π by $\pi[i] = s_i$ and the time spent in the i th state by $\delta(\pi, i)$. The time at which the execution enters state $\pi[i+1]$ is given by $\tau(\pi, i+1) = \sum_{j=0}^{j=i} \delta(\pi, j)$. The state of the execution at time t (if the sum of sojourn times in all states in the path exceeds t), denoted by $\pi(t)$, is the smallest i such that $t \leq \tau(\pi, i+1)$. We let $Path(s)$ be the set of executions starting at state s . We assume $Path(s)$ is a measurable set (in an appropriate σ -field) and has an associated probability measure.

2.2 Continuous Stochastic Logic

Continuous stochastic logic (CSL) is introduced in [1] as a logic to express probabilistic properties of continuous time Markov chains (CTMCs). In this paper we adopt a sublogic of CSL (excluding unbounded untils and stationary state operators) as in [16]. This logic excludes the steady-state probabilistic operators and the unbounded until operators. We next present the syntax and the semantics of the logic.

CSL Syntax

$$\begin{aligned}\phi ::= & \text{true} \mid a \in AP \mid \neg\phi \mid \phi \wedge \phi \mid \mathcal{P}_{\bowtie p}(\psi) \\ \psi ::= & \phi \mathcal{U}^{\leq t} \phi \mid \mathbf{X}\phi\end{aligned}$$

where AP is the set of atomic propositions, $\bowtie \in \{<, \leq, >, \geq\}$, $p \in [0, 1]$, and $t \in \mathbb{R}_{\geq 0}$. Here ϕ represents a *state* formula and ψ represents a *path* formula. The notion that a state s (or a path π) *satisfies* a formula ϕ is denoted by $s \models \phi$ (or $\pi \models \phi$), and is defined inductively as follows:

CSL Semantics

$$\begin{array}{lll}s \models \text{true} & s \models a & \text{iff } a \in AP(s) \\ s \models \neg\phi & \text{iff } s \not\models \phi & \\ s \models \mathcal{P}_{\bowtie p}(\psi) & \text{iff } \text{Prob}\{\pi \in Path(s) \mid \pi \models \psi\} \bowtie p & \\ \pi \models \mathbf{X}\phi & \text{iff } \tau(\pi, 1) < \infty \text{ and } \pi[1] \models \phi & \\ \pi \models \phi_1 \mathcal{U}^{\leq t} \phi_2 & \text{iff } \exists x \in [0, t]. (\pi(x) \models \phi_2 \text{ and } \forall y \in [0, x]. \pi(y) \models \phi_1)\end{array}$$

A formula $\mathcal{P}_{\bowtie p}(\psi)$ is satisfied by a state s if $\text{Prob}[\text{path starting at } s \text{ satisfies } \psi] \bowtie p$. To define probability that a path satisfies ψ we need to define a σ -algebra over the set of paths starting at s and a probability measure on the corresponding measurable space in a way similar to [5]. The path formula $\mathbf{X}\phi$ holds over a path if ϕ holds at the second state on the path. The formula $\phi_1 \mathcal{U}^{\leq t} \phi_2$ is true over a path π if ϕ_2 holds in some state along π at a time $x \in [0, t]$, and ϕ holds along all prior states along π . This can also be recursively defined as follows:

$$\begin{aligned}Sat(s_i \xrightarrow{t_i} \pi_{i+1}, \phi_1 \mathcal{U}^{\leq t} \phi_2) \\ = (t \geq 0) \wedge (Sat(s_i, \phi_2) \vee (Sat(s_i, \phi_1) \wedge Sat(\pi_{i+1}, \phi_1 \mathcal{U}^{\leq t-t_i} \phi_2)))\end{aligned}\quad (1)$$

where $Sat(s, \phi)$ (or $Sat(\pi, \psi)$) are the propositions that $s \models \phi$ (or $\pi \models \psi$). This definition will be used later to describe the algorithm for verification of $\phi_1 \mathcal{U}^{\leq t} \phi_2$ formula.

3 Algorithm

In what follows we say that $s \models_{\mathcal{A}} \phi$ if and only if our algorithm (denoted by \mathcal{A}) says that ϕ holds at state s . Since the algorithm is statistical, the decision made by the algorithm provides *evidence* about the actual fact. In our approach, we bound the strength of this evidence quantitatively by a number in $[0, 1]$ which gives the probability of making the decision given that the decision is actually incorrect. In statistics, this is called the p -value of the testing method. We denote it by α^3 and write $s \models \phi$ if the state s actually satisfies ϕ .

We assume that we are given a set of finite executions. The length of a finite execution path must be large enough so that all the bounded until formulas can be evaluated on that path. Given a set of sample execution paths starting at the initial state and a formula ϕ , the algorithm works recursively as follows:

```
verifyAtState( $\phi, s$ ){
    if cache contains ( $\phi, s$ ) return cache( $\phi, s$ );
    else if  $\phi = \text{true}$  then  $(z, \alpha) \leftarrow (1, 0.0)$ ;
    else if  $\phi = a \in AP$  then  $(z, \alpha) \leftarrow \text{verifyAtomic}(a, s)$ ;
    else if  $\phi = \neg\phi'$  then  $(z, \alpha) \leftarrow \text{verifyNot}(\neg\phi', s)$ ;
    else if  $\phi = \phi_1 \wedge \phi_2$  then  $(z, \alpha) \leftarrow \text{verifyAnd}(\phi_1 \wedge \phi_2, s)$ ;
    else if  $\phi = \mathcal{P}_{\bowtie p}(\psi)$  then  $(z, \alpha) \leftarrow \text{verifyProb}(\mathcal{P}_{\bowtie p}(\psi), s)$ ;
    store  $(s, \phi) \mapsto (z, \alpha)$  in cache;
    return  $(z, \alpha)$ ;
}
```

where *verifyAtState* returns a pair having 0, 1, or *undecided* corresponding to the cases $s \models_{\mathcal{A}} \phi$, $s \not\models_{\mathcal{A}} \phi$, or \mathcal{A} cannot decide respectively, as the first component, and p -value for this decision as the second component. To verify a system we check if the given formula holds at the initial state. Once computed, we store the decision of the algorithm for ϕ at state s in a cache to avoid recomputation. The result of our hypothesis testing can be shown to hold in presence of caching. This results in a significantly faster running time and a reduction in the sample set size. In the remainder of this section we define the various procedures *verifyAtomic*, *verifyAnd*, *verifyNot*, and *verifyProb* recursively.

The key idea of the algorithm is to statistically verify the probabilistic operator. We present the corresponding procedure *verifyProb* below.

3.1 Probabilistic Operator

We use statistical hypothesis testing [11] to verify a probabilistic property $\phi = \mathcal{P}_{\bowtie p}(\psi)$ at a given state s . Without loss of generality we show our procedure for $\phi = \mathcal{P}_{\geq p}(\psi)$. This is because, for the purpose of statistical analysis, $\mathcal{P}_{< p}(\psi)$ is essentially the same as $\neg\mathcal{P}_{\geq 1-p}(\psi)$ and $<$ (or $>$) is in effect the same as \leq (or \geq). Let p' be the probability that ψ holds over paths starting at s . We say that $s \models \mathcal{P}_{\geq p}(\psi)$ if and only if $p' \geq p$ and $s \not\models \mathcal{P}_{\geq p}(\psi)$ if and only if $p' < p$. We want to decide either $s \models \mathcal{P}_{\geq p}(\psi)$ or $s \not\models \mathcal{P}_{\geq p}(\psi)$. Accordingly we set up two experiments. In the first experiment, we use sample execution paths starting

³ This should not be confused with the Type I error which is also denoted by α

at s to test the *null hypothesis* $H_0: p' < p$ against the *alternative hypothesis* $H_1: p' \geq p$. In the second experiment, we test the *null hypothesis* $H_0: p' \geq p$ against the *alternative hypothesis* $H_1: p' < p$.⁴

Let the number of sample execution paths having a state s somewhere in the path be n . We can treat the portion of all these paths starting at s (suffix) as samples from $\text{Path}(s)$. Let X_1, X_2, \dots, X_n be a random sample having Bernoulli distribution with unknown parameter $p' \in [0, 1]$ i.e. for each $i \in [1, n]$, $\text{Prob}[X_i = 1] = p'$. Then the sum $Y = X_1 + X_2 + \dots + X_n$ has binomial distribution with parameters n and p' . We say that x_i , an observation of the random variable X_i , is 1 if the i^{th} sample execution path satisfies ψ and 0 otherwise. In the first experiment, we reject $H_0: p' < p$ and say $s \models_{\mathcal{A}} \mathcal{P}_{\geq p}(\psi)$ if $\frac{\sum x_i}{n} \geq p$ and calculate the p -value as $\alpha = \text{Prob}[s \models_{\mathcal{A}} \phi \mid s \not\models \phi] = \text{Prob}[Y \geq \sum x_i \mid p' < p]$. Note we do not know p' . Therefore, to calculate α we use p which is an upper bound for p' . If we are not able to reject H_0 in the first experiment then we do the second experiment. In the second experiment, we reject $H_0: p' \geq p$ and say $s \not\models_{\mathcal{A}} \mathcal{P}_{\geq p}(\psi)$ if $\frac{\sum x_i}{n} < p$ and calculate the p -value as $\alpha = \text{Prob}[s \not\models_{\mathcal{A}} \phi \mid s \models \phi] = \text{Prob}[Y < \sum x_i \mid p' \geq p]$. Thus, a smaller α represents a greater confidence in the decision of the algorithm \mathcal{A} .

3.2 Nested Probabilistic Operators

The above procedure for hypothesis testing works if the truth value of ψ over an execution path determined by the algorithm is the same as the actual truth value. However, in the presence of nested probabilistic operators in ψ , \mathcal{A} cannot determine the satisfaction of ψ over a sample path exactly. Therefore, in this situation we need to modify the hypothesis test so that we can use the inexact truth values of ψ over the sample paths.

Let the random variable X be 1 if a sample execution path π actually satisfies ψ in the system and 0 otherwise. Let the random variable Z be 1 for a sample execution path π if $\pi \models_{\mathcal{A}} \psi$ and 0 otherwise. In our procedure we cannot get samples from the random variable X ; instead our samples come from the random variable Z . Let X and Z have Bernoulli distributions with parameters p' and p'' respectively. Let Z_1, Z_2, \dots, Z_n be a random sample from the Bernoulli distribution with unknown parameter $p'' \in [0, 1]$. We say that z_i , an observation of the random variable Z_i , is 1 if *the algorithm says* that the i^{th} sample execution path satisfies ψ and 0 otherwise.

For the formula $\phi = \mathcal{P}_{\geq p}(\psi)$ we calculate regions of indifference, denoted by the fractions δ_1 and δ_2 , based on the algorithm's decision for the satisfaction of ψ over the different sample paths. Depending on the values of δ_1 and δ_2 we set up the two experiments. In the first experiment, we test the null hypothesis $H_0: p'' \leq p + \delta_1$ against the alternative hypothesis $H_1: p'' > p + \delta_1$. If we get $\frac{\sum z_i}{n} > p + \delta_1$ we reject H_0 and say $s \models_{\mathcal{A}} \mathcal{P}_{\geq p}(\psi)$ with p -value $\alpha = \text{Prob}[s \models_{\mathcal{A}} \phi \mid s \not\models \phi] = \text{Prob}[\sum Z_i > \sum z_i \mid p'' \leq p + \delta_1]$. If we fail to reject H_0 we

⁴ While handling nested probabilistic operators, these experiments will no longer be symmetric. Moreover, setting up these two experiments is an alternate way of getting at a conservative estimate of what Type II error (β value) may be.

go for the second experiment, in which the null hypothesis $H_0: p'' \geq p - \delta_2$ against the alternative hypothesis $H_1: p'' < p - \delta_2$. We reject H_0 and say that $s \not\models_{\mathcal{A}} \mathcal{P}_{\geq p}(\psi)$ if $\frac{\sum z_i}{n} < p - \delta_2$ and calculate the p -value as $\alpha = \text{Prob}[s \not\models_{\mathcal{A}} \phi \mid s \models \phi] = \text{Prob}[\sum Z_i < \sum z_i \mid p'' \geq p - \delta_2]$. Otherwise, we say the algorithm cannot decide.

We now show how to calculate δ_1 and δ_2 . Using the samples from Z we can estimate p'' . However, we need an estimation for p' in order to decide whether $\phi = \mathcal{P}_{\geq p}(\psi)$ holds in state s or not. To get an estimate for p' we note that the random variables X and Z are related as follows:

$$\text{Prob}[Z = 0 \mid X = 1] \leq \alpha' \quad \text{Prob}[Z = 1 \mid X = 0] \leq \alpha'$$

where α' is the p -value calculated while verifying the formula ψ . By elementary probability theory, we have

$$\text{Prob}[Z = 1] = \text{Prob}[Z = 1 \mid X = 0]\text{Prob}[X = 0] + \text{Prob}[Z = 1 \mid X = 1]\text{Prob}[X = 1]$$

Therefore, we can approximate $p'' = \text{Prob}[Z = 1]$ as follows:

$$\begin{aligned} \text{Prob}[Z = 1] &\leq \alpha'(1 - p') + 1.p' = p' + (1 - p')\alpha' \\ \text{Prob}[Z = 1] &\geq \text{Prob}[Z = 1 \mid X = 1]\text{Prob}[X = 1] \geq (1 - \alpha')p' = p' - \alpha'p' \end{aligned}$$

This gives the following range in which p'' lies:

$$p' - \alpha'p' \leq p'' \leq p' + (1 - p')\alpha'$$

Hence, $\text{Prob}[\sum Z_i > \sum z_i \mid p' \leq p] \leq \text{Prob}[\sum Z_i > \sum z_i \mid p' = p] \leq \text{Prob}[\sum Z_i > \sum z_i \mid p'' = p - \alpha'p]$ which gives $\delta_2 = \alpha'p$. Similarly, we get $\delta_1 = (1 - p)\alpha'$.

Note that the p -value obtained while verifying ψ over different sample paths are different. We take the worst or the maximum of all such p -values as α' . Moreover, since \mathcal{A} can say *true*, *false*, or *cannot decide*, note that the \mathcal{A} may not have a definite *true* or *false* answer along certain sample paths. For such paths the algorithm will assume the worst possible answer in the two experiments. For the first experiment, where we check whether $\frac{\sum z_i}{n} > p + \delta_1$, we take the answers for the sample paths for which \mathcal{A} cannot decide as *false* and the p -value as 0. For the second experiment we consider the answer for the undecided paths to be *true* and the p -value as 0. This allows us to obtain useful answers even when the sample does not have enough statistical evidence for the satisfaction of a subformula.

Thus we can define the procedure $\text{verifyProb}(\mathcal{P}_{\geq p}(\psi), s)$ as follows:

```
verifyProb( $\mathcal{P}_{\geq p}(\psi)$ ,  $s$ ){
     $zsum_{min} \leftarrow 0$ ;  $zsum_{max} \leftarrow 0$ ;  $\alpha' \leftarrow 0.0$ ;  $n \leftarrow 0$ ;
    for each sample path  $\pi$  starting at  $s$ {
         $(z, \alpha'') \leftarrow \text{verifyPath}(\psi, \pi)$ ;
        if  $z = \text{undecided}$  then  $\{zsum_{min} \leftarrow zsum_{min} + 1$ ;  $zsum_{max} \leftarrow zsum_{max} + 0\}$ 
        else  $\{zsum_{min} \leftarrow zsum_{min} + z$ ;  $zsum_{max} \leftarrow zsum_{max} + z\}$ 
         $\alpha' \leftarrow \max(\alpha', \alpha'')$ ;  $n \leftarrow n + 1$ ;
    }
}
```

```

    }
    if  $zsum_{max}/n > p + (1 - p)\alpha'$  then
        return  $(1, Prob[\sum Z_i > zsum_{max} \mid p'' = p + (1 - p)\alpha']);$ 
    else if  $zsum_{min}/n < p - p\alpha'$  then
        return  $(0, Prob[\sum Z_i < zsum_{min} \mid p'' = p - p\alpha']);$ 
    else return  $(undecided, 0.0);$ 
}
One can calculate  $Prob[\sum Z_i > zsum_{max} \mid p'' = p + (1 - p)\alpha']$  (or  $Prob[\sum Z_i > zsum_{min} \mid p'' = p - p\alpha']$ ) by noting that  $\sum Z_i$  has binomial distribution with parameters  $p + (1 - p)\alpha'$  (or  $p - p\alpha'$ ) and  $n$ .

```

3.3 Negation

For the verification of a formula $\neg\phi$ at a state s , we recursively verify ϕ at state s . If $s \models_{\mathcal{A}} \phi$ with p -value α we say that $s \not\models_{\mathcal{A}} \neg\phi$ with p -value $Prob[s \not\models_{\mathcal{A}} \neg\phi \mid s \models \phi] = Prob[s \models_{\mathcal{A}} \phi \mid s \not\models \phi] = \alpha$. Similarly, if $s \not\models_{\mathcal{A}} \phi$ with p -value α then $s \models_{\mathcal{A}} \neg\phi$ with p -value α . Otherwise, if \mathcal{A} cannot answer the satisfaction of ϕ at s , we say that \mathcal{A} cannot answer the satisfaction of $\neg\phi$ at s . Thus we can define *verifyNot* as follows:

```

verifyNot( $\neg\phi', s$ ){
     $(z, \alpha) \leftarrow verifyAtState(\phi', s);$ 
    if  $z = undecided$  then return  $(undecided, 0.0)$ ; else return  $(1 - z, \alpha);$ 
}

```

3.4 Conjunction

To verify a formula $\phi = \phi_1 \wedge \phi_2$ at a state s , we verify ϕ_1 and ϕ_2 at the state s separately. Depending on the outcome of the verification of ϕ_1 and ϕ_2 , \mathcal{A} decides for ϕ as follows:

1. If $s \models_{\mathcal{A}} \phi_1$ and $s \models_{\mathcal{A}} \phi_2$ with p -values α_1 and α_2 respectively, then $s \models_{\mathcal{A}} \phi$. The p -value for this decision is $Prob[s \models_{\mathcal{A}} \phi_1 \wedge \phi_2 \mid s \models \phi_1 \wedge \phi_2]$. Note that $s \not\models \phi_1 \wedge \phi_2$ holds in three cases, namely 1) $s \not\models \phi_1$ and $s \models \phi_2$, 2) $s \models \phi_1$ and $s \not\models \phi_2$, and 3) $s \not\models \phi_1$ and $s \not\models \phi_2$. Thus, the p -value for the decision of $s \models_{\mathcal{A}} \phi$ can be taken as the maximum of the p -values in the above three cases which is $\max(\alpha_1, \alpha_2)$.
2. If $s \not\models_{\mathcal{A}} \phi_1$ (or $s \not\models_{\mathcal{A}} \phi_2$) and either $s \models_{\mathcal{A}} \phi_2$ or \mathcal{A} cannot decide ϕ_2 at s (or either $s \models_{\mathcal{A}} \phi_1$ or \mathcal{A} cannot decide ϕ_1 at s), then $s \not\models_{\mathcal{A}} \phi$ and the p -value is $Prob[s \not\models_{\mathcal{A}} \phi_1 \wedge \phi_2 \mid s \models \phi_1 \text{ and } s \models \phi_2] = \alpha_1 + \alpha_2$ (or $\alpha_2 + \alpha_1$).
3. If $s \not\models_{\mathcal{A}} \phi_1$ and $s \not\models_{\mathcal{A}} \phi_2$ then $s \not\models_{\mathcal{A}} \phi_1 \wedge \phi_2$. The p -value for this decision is $Prob[s \not\models_{\mathcal{A}} \phi_1 \wedge \phi_2 \mid s \models \phi_1 \text{ and } s \models \phi_2] \leq \alpha_1 + \alpha_2$.
4. Otherwise, \mathcal{A} cannot decide ϕ .

Thus we can define the procedure *verifyAnd* as follows:

```

verifyAnd( $\phi_1 \wedge \phi_2, s$ ){
     $(z_1, \alpha_1) \leftarrow verifyAtState(\phi_1, s);$   $(z_2, \alpha_2) \leftarrow verifyAtState(\phi_2, s);$ 
}

```

```

    if  $z_1 = 1$  and  $z_2 = 1$  then return  $(1, \max(\alpha_1, \alpha_2))$ ;
    else if  $z_1 = 0$  and  $z_2 \neq 0$  then return  $(0, \alpha_1 + \alpha_2)$ ;
    else if  $z_1 \neq 0$  and  $z_2 = 0$  then return  $(0, \alpha_1 + \alpha_2)$ ;
    else if  $z_1 = 0$  and  $z_2 = 0$  then return  $(0, \alpha_1 + \alpha_2)$ ;
    else return  $(\text{undecided}, 0.0)$ ;
}

```

3.5 Atomic Proposition

In this simplest case, given $\phi = a$ and a state s , \mathcal{A} checks if $s \models_{\mathcal{A}} a$ or not by checking if $a \in AP(s)$ or not. If $a \in AP(s)$ then $s \models_{\mathcal{A}} \phi$ with p -value 0. Otherwise, $s \not\models_{\mathcal{A}} \phi$ with p -value 0.

```

verifyAtomic( $a, s$ ){
    if  $a \in AP(s)$  then return  $(1, 0.0)$ ; else return  $(0, 0.0)$ ;
}

```

3.6 Next

To verify a path formula $\psi = \mathbf{X}\phi$ over a path π , \mathcal{A} verifies ϕ at the state $\pi[1]$. If $\pi[1] \models_{\mathcal{A}} \phi$ with p -value α then $\pi \models_{\mathcal{A}} \psi$ with p -value α . Otherwise, if $\pi[1] \not\models_{\mathcal{A}} \phi$ with p -value α then $\pi \not\models_{\mathcal{A}} \psi$ with the same p -value. Thus we can define $verifyPath$ for $\mathbf{X}\phi$ as follows:

```

verifyPath( $\mathbf{X}\phi, \pi$ ){
    return verifyAtState( $\phi, \pi[1]$ );
}

```

3.7 Until

Let $\psi = \phi_1 \mathcal{U}^{\leq t} \phi_2$ be an until formula that we want to verify over the path π . We can recursively evaluate the truth value of the formula over the path π by following the recursive definition given by Equation 1. Given the truth value and the p -value for the formula $\phi_1 \mathcal{U}^{\leq t' - t_i} \phi_2$ over the suffix π_{i+1} , we can calculate the truth value of $\phi_1 \mathcal{U}^{\leq t'} \phi_2$ over the path π_i by applying the decision procedure for conjunction and negation. Observe that the recursive formulation in Equation 1 can be unrolled to obtain an equation purely in terms of conjunction and negation (and without any until formulas); it is this “unrolled” version that is used in the implementation for efficiency reasons.

4 Implementation and Performance

We have implemented the above algorithm as part of a prototype Java tool called VESTA. We successfully used the tool to verify several programs having a CTMC model.⁵ The performance of the verification procedure depends on the number of samples required to reach a decision with sufficiently small p -value. To get a smaller p -value the number of samples needs to be increased. We need a lot of samples only when the actual probability of a path from a state s satisfying a

⁵ We selected systems with CTMC model so that we can compare our results with that of existing tools.

formula ψ is very close to the threshold p in a formula $\mathcal{P}_{\bowtie p}(\psi)$ whose satisfaction we are checking at s .

To evaluate the performance and effectiveness of our implementation we did a few case studies. We mostly took the stochastic systems used for case studies in [15]. The experiments were done on a 1.2 GHz Mobile Pentium III laptop running Windows 2000 with 512 MB memory.⁶ We did not take into account the time required for generating samples: we assumed that such samples come from a running system. However, this time, as observed in some of our experiments, is considerably less than the actual time needed for the analysis. We generated samples of length sufficient to evaluate all the time-bounded until formulas. In all of our case studies we checked the satisfaction of a given formula at the initial state. We give a brief description of our case studies below followed by our results and conclusions. The details for the case studies can be obtained from <http://osl.cs.uiuc.edu/~ksen/vesta/>.

Grid World: We choose this case study to illustrate the performance of our tool in the presence of nested probabilistic operators. It consists of an $n \times n$ grid with a robot located at the bottom-left corner and a janitor located at the top-right corner of the grid. The robot first moves along the bottom edge of a cell square and then along the right edge. The time taken by the robot to move from one square to another is exponentially distributed. The janitor also moves randomly over the grid. However, either the robot or the janitor cannot move to a square that is already occupied by the other. The robot also randomly sends a signal to the base station. The underlying model for this example is a CTMC. The aim of the robot is to reach the top-right corner in time T_1 units with probability at least 0.9, while maintaining a minimum 0.5 probability of communication with the base station with periodicity less than T_2 units of time. This can be specified using the CSL formula $\mathcal{P}_{\geq 0.9}(\mathcal{P}_{\geq 0.5}(\text{true} \mathcal{U}^{\leq T_2} \text{communicate}) \mathcal{U}^{\leq T_1} \text{goal})$.

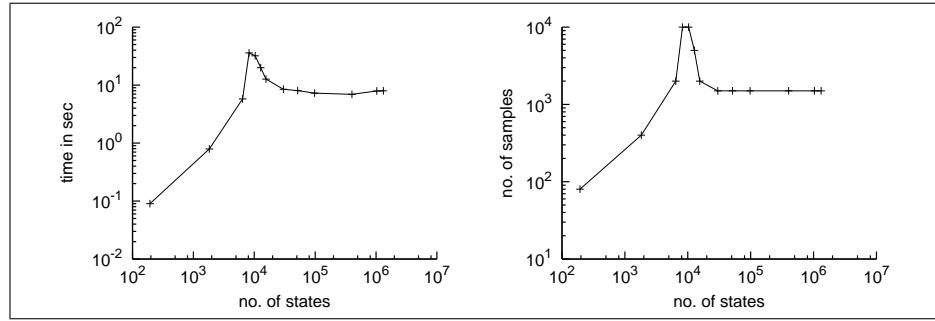


Fig. 1. Grid world: verification time and number of samples versus number of states.

We verified the CSL property for Grid World with $n \in [1, 100]$. The property holds only for $n \in [1, 13]$. The state space of the program is $\Theta(n^3)$. In Fig.1 we

⁶ [15] used a 500 MHz Pentium III. However, our performance gain due to the use of faster processor is more than offset by the use of Java instead of C.

plot the results of our experiment. The graph shows that for n closer to 13 the running time and the number of samples required increases considerably to get a respectable p -value of around 10^{-8} . This is because at $n = 13$ the probability that $\mathcal{P}_{\geq 0.5}(\text{true } \mathcal{U}^{\leq T_2} \text{ communicate}) \mathcal{U}^{\leq T_1} \text{ goal}$ holds over an execution path becomes very close to 0.9. We found that our graphs are similar to [15].

Cyclic Polling System: This case study is based on a cyclic server polling system, taken from [12]. The model is represented as a CTMC. We use N to denote the number of stations handled by the polling server. Each station has a single-message buffer and they are cyclically attended by the server. The server serves the station i if there is a message in the buffer of i and then moves on to poll the station $(i+1)$ modulo N . Otherwise, the server starts polling the station $i+1$ modulo N . The polling and service times are exponentially distributed. The state space of the system is $\Theta(N \cdot 2^N)$. We verified the property that “once a job arrives at the first station, it will be polled within T time units with probability at least 0.5.” The property is verified at the state in which all the stations have one message in their message buffer and the server is serving station 1. In CSL the property can be written as $(m_1 = 1) \rightarrow \mathcal{P}_{\geq 0.5}(\text{true } \mathcal{U}^{\leq T}(s = 1 \wedge a = 0))$, where $m_1 = 1$ means there is one message at station 1, and $s = 1 \wedge a = 0$ means that the server is polling station 1.

Tandem Queuing Network: This case study is based on a simple tandem queuing network studied in [9]. The model is represented as a CTMC which consists of a M/Cox₂/1-queue sequentially composed with a M/M/1-queue. We use N to denote the capacity of the queues. The state space is $\Theta(N^2)$. We verified the CSL property $\mathcal{P}_{< 0.5}(\text{true } \mathcal{U}^{\leq T} \text{ full})$ which states that the probability of the queuing network becoming full within T time units is less than 0.5.

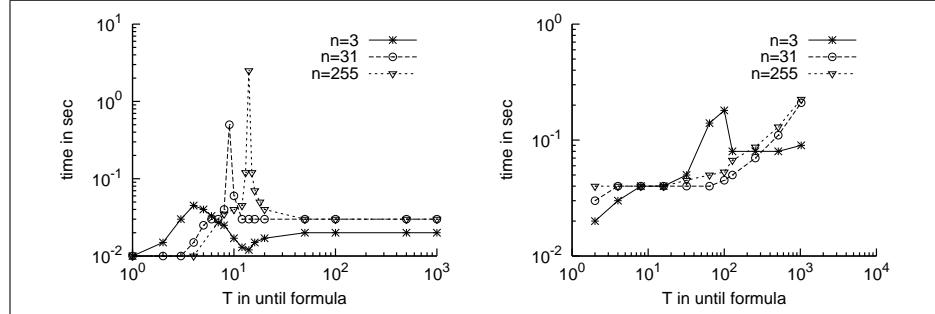


Fig. 2. Polling System and Tandem Queuing Network: Running time versus the parameter T in CSL formula.

The results of the above two case studies is plotted in Fig. 2. The characteristics of the graphs for both the examples are similar to that in [15]. However, while achieving a level of confidence around 10^{-8} , the running time of our tool *for these cases* is faster than the running time of the tool described in [15]. It is important to observe that, unlike the work of [15], VESTA cannot guarantee that the error of its probabilistic answer is bounded; the p -value computed depends on the specific sample.

We could not compare the number of samples for these case studies as they are not available from [15]; theoretically sequential hypothesis testing should require a smaller sample size than simple hypothesis testing to achieve the same level of confidence. While in our case studies we never faced a memory problem, we suspect that this may be a problem in very big case studies. We observed that, although the state space of a system may be large, the number of states that appeared in the samples may be considerably smaller. This gives us hope that our approach may work as well for very large-scale systems.

5 Conclusion and Future Work

We have presented a new statistical approach to verifying stochastic systems based on Monte Carlo simulation and statistical hypothesis testing. The main difference between our approach and previous statistical approaches is that we assume that the system under investigation is not under our control. This means that our algorithm computes on a fixed set of executions and cannot obtain samples as needed. As a consequence, the algorithm needs to check for the satisfaction of a property and compute the p -value of its tests. Since the sample may not provide sufficient statistical evidence to conclude the satisfaction or violation of a property, one technical challenge is to provide useful answers despite insufficient statistical evidence for the satisfaction of subformulas. We implemented our approach in Java and our experimental case studies have demonstrated that the running time of our tool is faster than previous methods for analyzing stochastic systems in at least some cases; this suggests that our method may be a feasible alternative.

Another important challenge is the amount of memory needed. Since we store all the sample executions, our method is memory intensive, and though we did not suffer from memory problems on the examples studied here, we suspect that it will be an issue when analyzing larger case studies. Hence, there is a need to design efficient data structures and methods to store and compute with a large set of sample executions. We suspect statistical hypothesis testing approaches (as opposed to sequential hypothesis testing approaches) might be extendible to check liveness for certain types of systems, possibly by extracting some additional information about the traces. Note that liveness properties are particularly complicated by the fact that the operators may be nested. We are currently exploring that direction. Finally, it would be interesting to apply statistical methods to analyze properties described in probabilistic logics other than CSL.

Acknowledgments

The work is supported in part by the DARPA IPTO TASK Award F30602-00-2-0586, the DARPA IXO NEST Award F33615-01-C-1907, the DARPA/AFOSR MURI Award F49620-02-1-0325, the ONR Grant N00014-02-1-0715, and the Motorola Grant MOTOROLA RPS #23 ANT. We would also like to acknowledge the contribution of Jose Meseguer to this research. Our work has benefitted considerably from stimulating discussions with him and from our many years of collaboration on probabilistic rewriting theories. We would like to thank Reza Ziaie for reviewing a previous version of this paper and giving us valuable feedback.

References

1. A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous-time Markov chains. In *8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102, pages 269–276. Springer, 1996.
2. R. Alur, C. Courcoubetis, and D. Dill. Model-checking for probabilistic real-time systems (extended abstract). In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming (ICALP'91)*, volume 510 of *LNCS*, pages 115–126. Springer, 1991.
3. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-checking continuous-time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.
4. C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Z. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*, pages 430–440. Springer, 1997.
5. C. Baier, J. P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time markov chains. In *International Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 146–161. Springer, August 1999.
6. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proceedings of 15th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS'95)*, volume 1026 of *LNCS*.
7. T. Herault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate Probabilistic Model Checking. In *Proceedings of Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 73–84. Springer, 2004.
8. H. Hermanns, J. P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov chain model checker. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'00)*, pages 347–362, 2000.
9. H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi-terminal binary decision diagrams to represent and analyse continuous-time markov chains. In *Proceedings of 3rd International Workshop on the Numerical Solution of Markov Chains (NSMC'99)*, 1999.
10. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
11. R. V. Hogg and A. T. Craig. *Introduction to Mathematical Statistics*. Macmillan, New York, NY, USA, fourth edition, 1978.
12. O. C. Ibe and K. S. Trivedi. Stochastic petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.
13. M. Z. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker, 2002.
14. M. Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Verifying quantitative properties of continuous probabilistic timed automata. In *International Conference on Concurrency Theory (CONCUR'00)*, volume 1877 of *LNCS*, pages 123–137. Springer, 2000.
15. H. L. S. Younes, M. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking: An empirical study. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*. Springer, 2004.
16. H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 223–235. Springer, 2002.

Maximal Clique Based Distributed Group Formation for Autonomous Agent Coalitions

Predrag Totic*, Gul Agha
Open Systems Laboratory, Department of Computer Science
University of Illinois at Urbana-Champaign, USA
{p-totic, agha}@cs.uiuc.edu

Abstract

We present herein a fully distributed algorithm for group or coalition formation among autonomous agents. The algorithm is based on a distributed computation of maximal cliques (of up to pre-specified size) in the underlying graph that captures the interconnection topology of the agents. Hence, given the current configuration of the agents, the groups that are formed are characterized by a high degree of connectivity, and therefore high fault tolerance with respect to node and link failures. We also briefly discuss how our basic algorithm can be adapted in various ways so that the formed groups satisfy the requirements ("goodness" criteria) other than mere strong inter-group communication connectivity. We envision various variants of our basic algorithm to prove themselves useful subroutines in many multi-agent system and ad hoc network applications where the agents may repeatedly need to form temporary groups or coalitions in an efficient, fully distributed and online manner.

Keywords: distributed algorithms, distributed group formation, multi-agent systems, autonomous agents, agent coalitions

is either no central control, or at best only a limited central control, and the individual agents have to both think and act locally, genuinely distributed algorithms are needed for the agents to effectively coordinate with one another. MAS pose a number of challenges to a distributed algorithm designer; most of these challenges are related to various aspects of the agent coordination problem [1, 18]. In order to be able to effectively coordinate, agents need to be able to *reach consensus (or agreement)* on various matters of common interest. Two particularly prominent and frequently encountered consensus problems are those of *leader election* (e.g., [6, 15]) and *group formation*.

Group formation is an important issue in distributed systems in general, and MAS in particular. Given a collection of computational and communicating agents, the goal in distributed group formation is that these agents, based on their local knowledge only, decide how to effectively split up into several groups, so that each agent knows what group(s) it belongs to.

There are several critical issues that a MAS designer needs to address in the context of (distributed) group formation. First, what is the right notion of a group in a given setting? That is, how is the quality of a particular group configuration measured, so that one can say that one grouping of agents into coalitions is better for that agent and/or for the whole system than another? Second, a distributed group formation mechanism - that is, a distributed algorithm that enables agents to effectively form groups or coalitions - needs to be provided. Third, groups and each agent's knowledge about its group membership need to be maintained and, when needed, appropriately updated. Another important issue is whether the groups are allowed to overlap, i.e., whether an agent is allowed to simultaneously belong to two or more groups. Variants of these basic challenges

1 Introduction and Motivation

Multi-agent systems (MAS) are characterized, among other properties, by (i) considerable degree of autonomy of individual computing entities or processes (agents) and, at the same time, the fact that (ii) each agent has a local, that is, in general, incomplete and imperfect "picture of the world". Since in MAS there

*Contact author. Phone numbers: 217-244-1976 (work), 217-390-6515 (cell); fax: 217-333-9386 (for calls/faxes from within the U.S.). Mailing address: Department of Computer Science, 1334 Siebel Center, 201 N. Goodwin, Urbana, IL 61801 USA

are quite common in the MAS literature; indeed, these challenges have arisen in our own recent and current work on parametric models and a scalable simulation of large scale ($10^3 - 10^4$ agents) ensembles of autonomous unmanned vehicles on a multi-task mission [5, 16, 17].

Herein, however, we restrict our attention to the second issue mentioned above. We propose a particular mechanism (distributed algorithm) for an effective coalition formation that ensembles of autonomous agents can use as one of their basic coordination subroutines. A need for a distributed, online and efficient group formation may arise due to a number of different factors, such as the geographical dispersion of the agents, heterogeneity of tasks and their resource requirements, heterogeneity of agents' capabilities, and so on. While for small- and medium-scale systems of robots or unmanned vehicles a fully or partially centralized approach to group formation and maintenance may be feasible or even optimal, large scale systems (with the number of agents of orders of magnitude as in [5] or higher) seem to require a fully distributed approach. That is, the agents need to be able to self-organize into coalitions, and quickly reach a consensus on who is forming a coalition with whom. The algorithm we present herein is an attempt to address these challenges shared by many large-scale MAS applications.

We remark that the group formation and the leader election problems are often inter-related. In particular, at least three general approaches to the combined problem of both forming groups and electing group leaders can be identified. One approach is, that groups are formed by an authority from "the outside", and then the agents within each thus formed group are to reach consensus on who is to be the group leader. That is, only the leader election part is distributed. Distributed leader election (once the group structure is already in place) has been extensively studied in various distributed system models (e.g., [6]).

Another approach is to first select leaders (possibly by appointing them from the outside), and then let these leaders agree with one another on how to assign the rest of the agents to groups "belonging" to different leaders. For very large scale systems and dense interconnection topologies, this is likely the most feasible approach due to its scalability.

The third possibility of how to attack the joint group-formation-and-leader-election problem is that the agents in the ensemble self-organize and form groups first, and then, as in the first scenario, agents within each group decide on their leaders. That is, group formation precedes leader election, and both are done in a genuinely distributed manner. We argue that

this approach is scalable for large scale MAS provided that the interconnection topology of the ad hoc network the agents are forming (i) is relatively sparse, and (ii) does not change too rapidly. While our algorithm in Section 4 is generic, in designing it we were admittedly motivated by the autonomous unmanned vehicle applications - more specifically, by the micro unmanned aerial vehicles deployed over a sizable geographical area (see [5, 17]).

2 Group Formation in Multi-Agent Systems

Large ensembles of autonomous agents provide an important class of examples where the agents' capability to coordinate and, in particular, to self-organize into groups or coalitions, is often of utmost importance for such systems to be able to accomplish their tasks, or, in some cases, even to merely survive. One can distinguish between two general, broad classes of such autonomous agents. One is the class of agents employed in *distributed problem solving*. The agents encountered in distributed problem solving (DPS) typically share their goal(s). For instance, DPS agents most often have a joint utility function that they all wish to maximize as a team, without any regard to (or sometimes even a notion of) individual payoffs. This joint utility or, more generally, the goal or set of goals assigned to DPS agents, is usually provided by their designer. However, it may not be always feasible - or even possible - that the designer explicitly provide, for instance, how are the agents to divide-and-conquer their tasks and resources, how are they to form groups and elect leaders of those groups, etc. Due to scalability, incomplete *a priori* knowledge of the environments these agents may encounter, and possibly other considerations, instead of "hard-wiring" into his DPS agents explicitly how are the agents to be coordinated, the system designer may choose only to enable the agents with the basic coordination primitives, and leave to them to self-organize and coordinate as the situation may demand. Hence, in many situations the DPS agents may be required to be able to effectively form groups or coalitions in a fully distributed manner.

The second basic type of agents, the *self-interested agents*, are a kind of agents that do not share their goals (and, indeed, need not share their designer). In contrast to the DPS agents, each self-interested agent has its own agenda (e.g., an individual utility function it is striving to maximize), and no altruistic incentives to cooperate with other agents. Even such self-interested, goal-driven or individual-utility-driven agents, while in

essence selfish, may nonetheless still need to cooperatively coordinate and collaborate with each other in many situations.

One class of examples are those agents (such as, e.g., autonomous unmanned vehicles) that, if they do not coordinate in order to resolve possible conflicts, they risk mutual annihilation. Another class of examples are the agents with bounded resources: individually, an agent may lack resources to accomplish any of its desired tasks - yet, if this agent forms a coalition with one or more other agents, the combined resources and joint effort of all agents in such a coalition may provide utility benefits to everyone involved.

For these reasons, group formation and coalition formation are of considerable interest for many different kinds of autonomous agents and multi-agent systems, and, among other, even in those multi-agent systems where the agents do not share a global utility function, and where each agent generally acts selfishly. In particular, efficient fully distributed algorithms for effective group formation are needed. Such algorithms should use only a few communication rounds among the agents, place a very modest computational burden on each agent, and ensure that a distributed consensus among the agents - that is, in the end, who is forming a group with whom - is effectively, reliably and efficiently reached.

We propose herein one such class of distributed algorithms. We describe in some detail the generic, distributed max-clique-based group formation algorithm in Section 4. Various variants of this basic max-clique-based group formation algorithm can be designed to suit the needs of various types of agents, such as, e.g., the classical DPS agents, the self-interested agents, and the resource-bounded agents.

First, we discuss a generic distributed group formation algorithm based on the idea that an agent (node) would prefer to join a group with those agents that it can communicate with directly, and, moreover, where every member of such a potential group can communicate with any other member directly. That is, the preferable groups (coalitions) are actually (maximal) cliques. It is well-known that finding a maximal clique in an arbitrary graph is **NP**-complete in the centralized setting [3,4]. This implies the computational hardness that, in general, each node faces when trying to determine maximal clique(s) it belongs to. However, if the degree of a node (that is, its number of neighbors in the graph) is small (in particular, if it is $O(1)$), then finding all maximal cliques this node belongs to is computa-

tionally feasible. If one cannot guarantee that (or does not know if) all the nodes in a given underlying MAS interconnection topology are of small degree, then one has to impose additional constraints in order to ensure that the agents (“nodes”) are not attempting to solve an infeasible problem. In particular, we shall additionally require herein that the cliques to be considered - that is, the possible coalitions to be formed - be of up to a certain pre-specified maximum size. Once such coalitions are formed, being cliques, they can be expected to be relatively robust with respect to the subsequent node or link failures in the system.

Second, we outline how this basic maximal clique based algorithm can be fine-tuned so that the formed coalitions are of good quality with respect to criteria other than the mere robustness with respect to link or node failures. In particular, we indicate how, in multi-agent, bounded resource, multi-task environments (as in, e.g., [17]), the maximal clique algorithm can be adjusted so that each agent strives to join a group such that the joint resources of all the agents in the group match this particular agent’s needs in additional resources. Such a choice of the group (coalition) would enable the agent to now be able to complete some of the tasks that this agent would not be able to complete with its own resources alone.

A variety of coalition formation mechanisms and protocols have been proposed in the MAS literature both in the context of DPS agents that are all sharing the same goal (as, e.g., in [13]) and in the context of self-interested agents where each agent has its own individual agenda (as, e.g., in [12, 22]). In particular, the problem of distributed task or resource allocation, and how is this task allocation coupled to what coalition structures are (most) desirable in a given scenario [13], are the issues of central importance in our own work on MAS in bounded resource multi-task environments [5, 16, 17]. These considerations have in part also motivated our work, and especially the extensions of the basic max clique based group formation algorithm, where the cost functions or coalition quality metrics, other than each coalition’s interconnectivity alone, are used to determine which coalition(s) is (are) most preferred by which agent¹. However, while in [13] all agents share all of their goals, we assume herein that each agent is self-interested and, in particular, each agent therefore has its own preference ordering on the coalitions that it may consider joining.

Another body of MAS literature related to our work on modeling and simulation of large scale ensembles of

¹In the present work, due to space constraints, this issue of how to generalize our algorithm to more general and specifically more resource-oriented coalition cost functions will be only briefly touched upon in Section 5. However, such generalizations, and their implementation in our UAV simulator [5], are a high priority agenda in our future work.

UAVs [5, 16, 17] casts the distributed resource allocation problems into the distributed constraint satisfaction (DCS) terms [7, 8]. The importance of DPS in MAS in general is discussed in [21]. However, discussing DCS based approaches to distributed resource or task allocation and coalition formation is beyond the scope of this paper.

3 Problem Statement and Main Assumptions

Our goal is to design a generic, fully distributed, scalable and efficient algorithm for ensembles of autonomous agents to use as a subroutine (or a coordination strategy) with a purpose of efficiently forming (temporary) groups or coalitions. The proposed algorithm is a graph algorithm. Each agent is a node in the graph. As for the edges, the necessary requirement for an edge between two nodes to exist is that the two nodes be able to directly communicate with one another at the time our distributed group formation subroutine is called².

The basic idea is to efficiently partition this graph into (preferably, maximal) cliques of nodes. These maximal cliques may also need to satisfy some additional criteria in order to form temporary coalitions. These coalitions are then maintained until they are no longer useful or meaningful. For instance, the coalitions should be transformed (or else simply dissolved) when the interconnection topology of the underlying graph considerably changes, either due to the agents' mobility, or because many old links have died out and perhaps many new, different links have formed, and the like. Another possible reason to abandon the existing coalition structure is when the agents determine that the coalitions have accomplished the set of tasks that these coalitions were formed to address. Thus, in an actual MAS application, the proposed group formation algorithm may need to be invoked a number of times as a coordination subroutine.

The algorithm is sketched in the next section. For this algorithm to work, the following basic assumptions need to hold:

- agents communicate with one another by exchanging messages either via local broadcasts, or in a peer-to-peer fashion;

²We point out that this definition of the graph edges can be made tighter by imposing additional requirements, such as, e.g., that the two agents to be connected by a link also be compatible, for instance, in terms of their capabilities, or that they each provide some resource(s) that the other agent needs, or the like.

³As this requirement is still quite restrictive, and considerably limits the robustness of our algorithm, we will try to relax this assumption in our future work.

⁴...at least for this particular invocation of the group formation subroutine.

- communication bandwidth availability is assumed not to be an issue;

- each agent has a sufficient local memory to be able to store all the information that it receives from other agents; this information is cf. made of the lists of neighboring nodes and of the coalitions proposed to this agent - see *Section 4*;

- communication is reliable during the group formation, but, once the groups are formed, this need no longer hold³;

- each agent has (or else can efficiently obtain) a reliable knowledge of what other agents are within its communication range;

- each agent, or node, has a unique global identifier (heretofore referred to as 'UID'), and it knows its UID;

- there is a total ordering, \prec , on the set of UIDs, and each agent knows this ordering \prec ;

- communication ranges of different agents are identical - in particular, if agent A can communicate messages to agent B , then also B can communicate messages to A .

On the other hand, an agent need not a priori know UIDs of other agents, or, indeed, how many other agents are present in the system.

In addition to its globally unique identifier (UID), which we assume is a positive integer, each agent has two local flags that it uses in communication with other agents. One of the flags is the binary "decision flag", which indicates whether or not this agent has already joined some group (coalition). Namely, $decision_flag \in \{0, 1\}$, and the value of this flag is 0 as long as the agent still has not irrevocably committed to what coalition it is joining. Once the agent makes this commitment, it updates the decision flag to 1 and broadcasts this updated flag value to all its neighbors (see below). That is, as long as the decision flag is 0, the agent's proposals of what group it would like to form or join are only tentative. Once the decision flag becomes 1, however, this indicates that the agent has made a committed choice of which coalition to join - and this selection is final⁴.

The second flag is the "choice flag", which is used to indicate to other agents, how "happy" the agent is with its current tentative choice or proposal of the group to be formed. That is, the choice flag indicates the level of agent's urgency that its proposal for a particular coalition to be formed be accepted by the neighbors in the interconnection topology to whom this proposal is sent.

It takes values $choice_flag \in \{0, 1, 2, 3\}$. When an agent is sending just its neighborhood list (at round one of the algorithm - see next section), the value of this flag is 3. Else, if the current choice of a coalition C_i proposed by agent i has equally preferable alternatives⁵, then i sets $choice_flag(i) \leftarrow 2$. If i has other available choices of groups it would like to join, yet each of these alternative choices is strictly less preferable to i than the current proposal (e.g., if each other possible group is a maximal clique of strictly smaller size than the maximal clique $C_i^{current}$ that is the agent i 's current proposal), then $choice_flag(i) \leftarrow 1$. Finally, if i has no other alternatives for a coalition proposal (beside the trivial coalition $\{i\}$), then $choice_flag(i) \leftarrow 0$. Hence, an agent whose current value of the choice flag is equal to 0 is quite desperate that its proposal of a coalition be accepted by those neighboring agents to whom the proposal is directed. In contrast, an agent whose current value of the choice flag is equal to 2 has some equally good alternative choices and can therefore change its mind without being penalized in terms of having to settle for a less preferable coalition.

4 Maximal Clique Based Distributed Group Formation

Now that the assumptions have been made explicit and the notation has been introduced, the stage is set for presenting our distributed maximal clique based coalition formation algorithm. The algorithm proceeds in five major stages, as follows:

Stage 1:

Set $counter \leftarrow 1$.

Each node (in parallel) broadcasts a tuple to all its immediate neighbors. The entries in this tuple are (i) the node's UID, (ii) the node's list of (immediate) neighbors, $L(i)$, (iii) the value of the choice flag, and (iv) the value of the decision flag.

WHILE (the agreement has not been reached) DO

Stage 2:

Each node (in parallel) computes the overlaps of its neighborhood list with the neighborhood lists that it has received from its neighbors, $C(i, j) \leftarrow L(i) \cap L(j)$. Repetitions (if any) among this neighborhood list intersections are deleted; the remaining intersections are ordered with respect to the list size (the ties, if any, are broken arbitrarily), and a new (ordered) collection of

⁵In the max. clique context, this means, if there are two or more maximal cliques of the same size, one of which is chosen by an appropriate tie-breaker. This idea readily generalizes, as long as each agent has a partial order of preferences over all the possible coalitions that include this agent.

these intersection lists (heretofore referred to simply as 'lists') is then formed.

If $counter > 1$ then:

Each node looks for information from its neighbors, whether they have joined a group "for good" during the previous round. Those neighbors that have (i.e., whose $decision_flag = 1$), are deleted from the neighborhood list $L(i)$; the intersection lists $C(i, j)$ are also updated accordingly, and those $C(i, k)$ for which k is deleted from the neighborhood list $L(i)$ are also deleted.

Stage 3:

Each node (in parallel) picks one of the most preferable lists $C(i, j)$; let $C(i) \leftarrow chosen [C(i, j)]$. If the list size is the main criterion, then this means, that one of the lists of maximal length is chosen. The value of the choice flag is set, based on whether an agent has other choices of lists as preferable as the chosen clique, and, if not, whether there are any other still available choices at all.

Stage 4:

Each node (in parallel) sends its tuple with its UID, the tentatively chosen list $C(i)$, the value of the choice flag, and the value of the decision flag, to all its neighbors.

Stage 5:

Each node i (in parallel) compares its chosen list $C(i)$ with lists $C(j)$ received from its neighbors. If a (relatively small, of size not exceeding k) clique that includes the node i exists, and all members of this clique have selected it at this stage as their current group or coalition of choice (that is, if $C(i) = C(j)$ for all $j \in C(i)$), this will be efficiently recognized by the nodes forming this clique. The decision flag of each node $j \in C(i)$ is set to 1, a group is formed, and this information is broadcast by each node in the newly formed group to all of its neighbors. Else, if no such agreement is reached, then agent i , based on its UID and priority, and its current value of the $choice_flag$, either does nothing, or else changes its mind about its current group of choice, $C(i)$ (the latter being possible only if $choice_flag > 0$, meaning that there are other choices of $C(i)$ that have not been tried out yet that are still available to agent i).

$counter \leftarrow counter + 1$;

END DO [* end of WHILE loop *]

If $round > 1$ then, at Stage 2, each node looks for the information from its neighbors to find out if any of them have joined a group in the previous round. For those nodes that have (i.e., whose decision flag

$dec = 1$), each node neighboring any such already committed node deletes this committed node from its neighborhood list $L(i)$, updates all $C(i,j)$ that remain, and selects its choice of $C(i)$ based on the updated collection of group choices $\{C(i,j)\}$. That is, now all those nodes that have already made their commitments and formed groups are not “in the game” any more, and are therefore deleted from all remaining agents’ neighborhood lists as well as the tentative choices of coalitions. (Of course, the only coalition a committed agent is *not* deleted from at this stage is *the coalition* that this agent has just joined).

There are some more details in the algorithm that we leave out for the space constraint reasons. One important technicality is that, in order to ensure that the algorithm always avoids to cycle, once an agent changes its mind about the preferred coalition $C(i)$, it is not allowed through the remaining rounds of the algorithm to go back to its old choice(s). Once no other choices are left, this particular agent sticks to its current (and the only remaining) choice, and waits for other agents to converge to their choices. It can be shown that this ensures ultimate convergence to a coalition structure that all agents agree to. That is, under the assumptions stated in the previous section, the agents will reach consensus on the coalition structure after a finite number of rounds inside the WHILE loop (see also *Appendix*). Moreover, if the maximum size of any $L(i)$ is a (small) constant, then the convergence is fast.

5 Discussion and Extensions

We have outlined a fully distributed algorithm for group or coalition formation based on maximal cliques. This algorithm will be efficient when the underlying graph is relatively sparse, and, in particular, when the sizes of all maximal cliques are bounded by a small constant $k = O(1)$. When this is not the case (or when it cannot be guaranteed to always hold), appropriate restrictions can be imposed “from the outside” to ensure that the algorithm converges, and rapidly. For example, for each node i , a range of possible values (UIDs) of those nodes that the node i is allowed to communicate and form coalitions with can be appropriately specified.

Once the groups are formed, these groups will be tight (as everyone in the group can communicate with everyone else), and, in nontrivial cases, therefore as robust as possible for a given number of group elements with respect to either node or link failures. This is a highly desirable property involving coalitions or teams of agents (robots, autonomous unmanned vehicles, etc.)

operating in environments where both the agent failures and the agent-to-agent communication link failures can be expected.

The proposed algorithm can be used as a subroutine in many multi-agent system scenarios where, at various points in time, the system needs to reconfigure itself, and the agents need to form new coalitions (or transform the existing ones) in a fully distributed manner, where each agent would join an appropriate (new) coalition because the agent finds this to be in its individual best interest, and where it is important to agents to form and agree on these coalitions efficiently, rather than wasting too much time and other resources on (possibly lengthy) negotiation.

This algorithm as a coordination subroutine in MAS applications can be expected to be useful only when the time scale of significant changes in the inter-agent communication topology is much coarser than the time scale for the coalitions of agents, first, to form according to the algorithm, and, second, once formed, to accomplish something useful in terms of agents’ ultimate goals. We are currently exploring using some version of this algorithm as a coordination strategy subroutine in a scalable software simulation of a system of unmanned autonomous aerial vehicles (UAVs) on a multi-task mission.

To be useful in various MAS applications, such as the above-mentioned UAV simulation, the proposed generic algorithm can be appropriately fine-tuned, so that the coalitions are formed that satisfy quality criteria other than robustness with respect to agent or communication link failures.

Let us assume there is an ensemble of self-interested agents moving around and about in an environment, searching for tasks and possible coalition partners and/or other external resources, and trying to service as many tasks as possible. Each task has a certain value to each agent: an agent increases its utility by servicing some task, and consuming this task’s value [16]. However, servicing different tasks requires resources, and an agent may lack sufficient resources to be able to service tasks it would like to complete. Hence, such an agent will have an egotistic incentive [2] to cooperatively coordinate with other agents - and, in particular, to try to form a group or a temporary coalition with other agents. The preferred partners in such a coalition would be those agents that would provide sufficient resources for the desired tasks to be completed. Hence, in the algorithm above, a refinement of the criterion of “goodness” (quality) of different groups that can be formed is needed. So, for example, at Stage 3, among the available lists of intersections $C(i,j)$, the agent i may want to choose one where the sum of resources of

all the agents in this list is equal to, or exceeds (but preferably by not too much) the resource requirements of the particular task the agent i desires to service⁶.

6 Concluding Remarks

We have proposed herewith a generic algorithm for distributed group formation based on maximal cliques of modest sizes. Among the existing distributed group formation algorithms, we argue that our algorithm is particularly suitable for dynamic coalition formation and transformation in multi-agent systems whose underlying graph structures (“topologies”) change frequently, yet not too rapidly. In particular, we find this algorithm, or its appropriately fine-tuned variants, to be a potentially very useful subroutine in many multi-agent system applications, where the interconnection topology of the agents often changes so that the system needs to dynamically reconfigure itself, yet where these topology changes are at a time scale that allows agents to (i) form coalitions, and (ii) do something useful while participating in such coalitions, before the underlying communication topology of the system changes so much as to render the formed coalitions either obsolete or ineffective. We intend to explore and test the applicability and practical usefulness of the proposed algorithm as a coordination strategy for various MAS applications in general, and in the context of our scalable simulation of autonomous unmanned vehicles on a complex, multi-task mission [5, 16, 17], in particular.

Acknowledgment: Many thanks to Nirman Kumar and Reza Ziae (Open Systems Laboratory, UIUC) for many useful discussions. This work was supported by the *DARPA IPTO TASK Program* under the contract *F30602-00-2-0586*.

Bibliography

- [1] N. M. Avouris, L. Gasser (eds.), “Distributed Artificial Intelligence: Theory and Praxis”, Euro Courses Comp. & Info. Sci. vol. 5, Kluwer Academic Publ., 1992
- [2] D. H. Cansever, ”Incentive Control Strategies For Decision Problems With Parametric Uncertainties”, Ph.D. thesis, Univ. of Illinois Urbana-Champaign, 1985
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, “Introduction to Algorithms”, MIT Press, 1990
- [4] M. R. Garey, D. S. Johnson, “Computers and Intractability: a Guide to the Theory of NP-completeness”, W. H. Freedman & Co., New York, 1979
- [5] M. Jang, S. Reddy, P. Tasic, L. Chen, G. Agha, “An Actor-based Simulation for Studying UAV Coordination”, Proc. 15th Euro. Symp. Simul. (ESS 2003), Delft, The Netherlands, October 2003
- [6] N. Lynch, “Distributed Algorithms”, Morgan Kaufmann Publ., Wonderland, 1996
- [7] P. J. Modi, H. Jung, W. Shen, M. Tambe, S. Kulkarni, “A dynamic distributed constraint satisfaction approach to resource allocation”, in “Principles and Practice of Constraint Programming”, 2001
- [8] P. J. Modi, W. Shen, M. Tambe, M. Yokoo, “An asynchronous complete method for distributed constraint optimization”, Proc. AAMAS 2003
- [9] J. von Neumann, O. Morgenstern, “Theory of Games and Economic Behavior”, Princeton Univ. Press, 1944
- [10] J. Rosenschein, G. Zlotkin, “Rules of Encounter: Designing Conventions for Automated Negotiations among Computers”, The MIT Press, Cambridge, Massachusetts, 1994
- [11] S. Russell, P. Norvig, “Artificial Intelligence: A Modern Approach”, 2nd ed., Prentice Hall Series in AI, 2003
- [12] O. Shehory, S. Kraus, “Coalition formation among autonomous agents: Strategies and complexity”, Proc. MAAMAW’93, Neuchatel, 1993
- [13] O. Shehory, S. Kraus, “Task allocation via coalition formation among autonomous agents”, Proc. 14th IJCAI-95, Montreal, August 1995
- [14] R. G. Smith, “The contract net protocol: high-level communication and control in a distributed problem solver”, IEEE Trans. on Computers, 29 (12), 1980
- [15] G. Tel, “Introduction To Distributed Algorithms”, 2nd ed., Cambridge Univ. Press, 2000
- [16] P. Tasic, M. Jang, S. Reddy, J. Chia, L. Chen, G. Agha, “Modeling a System of UAVs on a Mission”, Proc. SCI 2003 (invited session), Orlando, Florida, July 2003
- [17] P. Tasic, G. Agha, “Modeling Agents’ Autonomous Decision Making in Multiagent, Multitask Environments”, Proc. 1st Euro. Workshop MAS (EUMAS 2003), Oxford, England, 2003
- [18] G. Weiss (ed.), “Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence”, The MIT Press, Cambridge, Massachusetts, 1999
- [19] M. Wooldridge, N. Jennings, “Intelligent Agents: Theory and Practice”, Knowledge Engin. Rev., 1995
- [20] M. Yokoo, K. Hirayama, “Algorithms for Distributed Constraint Satisfaction: A review”, AAMAS, Vol. 3, No. 2, 2000
- [21] M. Yokoo, “Distributed Constraint Satisfaction: Foundation of Cooperation in Multi-agent Systems”, Springer, 2001
- [22] G. Zlotkin, J.S. Rosenschein, “Coalition, cryptography and stability: Mechanisms for coalition formation in task oriented domains”, Proc. AAAI’94, Seattle, Washington, 1994

⁶At this stage, however, we need to do more work on formalizing some of these variants of the algorithm, and, in particular, determine the criteria that would still ensure, under general assumptions similar as before, that the agents would efficiently converge to an agreement on the coalition structure.

Appendix: Pseudo-code for Max-Clique-Based Distributed Group Formation

Notation:

$i :=$ the i -th agent (node) in the system (say, $i = 1, \dots, n$)
 $V(i) :=$ the i -th node's UID
 $N(i) :=$ the list of neighbors of node i
 $L(i) :=$ the extended neighborhood list (i.e., $L(i) = N(i) \cup \{i\}$)
 $C(i, j) = L(i) \cap L(j)$
 $C(i) :=$ the group of choice of node i at the current stage (i.e., one of the $C(i, j)$'s)
 $choice(i) :=$ the choice flag of node i
 $dec(i) :=$ the decision flag of node i

Maximal Clique Based Distributed Group Formation Algorithm:

Step 1:

```
DOALL  $i = 1..n$  (in parallel, i.e., each node  $i$  carries the steps below locally)
    send  $[V(i), L(i), choice(i) = 3, dec(i) = 0]$  to each of your neighbors
END DOALL
```

WHILE (not all agents have joined a group) DO

Step 2:

```
DOALL  $i = 1..n$ 
    FOR all  $j \in N(i)$  DO [ $\star$  check if  $dec(j) = 1$   $\star$ ]
        if  $dec(j) == 1$  then delete  $j$  from  $N(i), L(i)$ , and  $C(i, j')$ ,  $\forall j' \in N(i) - \{j\}$ 
    END DO [ $\star$  end of FOR loop  $\star$ ]
    FOR all  $j \in N(i)$  DO [ $\star$  FOR all remaining (undeleted) indices  $j$   $\star$ ]
        compute  $C(i, j) \leftarrow L(i) \cap L(j)$ 
    END DO [ $\star$  end of FOR loop  $\star$ ]
END DOALL
```

Step 3:

```
DOALL  $i = 1..n$ 
    pick  $C(i, j)$  s.t.  $|C(i, j)|$  is of max. size (not exceeding the pre-specified threshold,  $k$ ):
         $C(i) \leftarrow C(i, j);$ 
    if more than one such choice, set  $choice(i) \leftarrow 2$ ;
    else (if there are other choices  $C(i, j')$  but only of smaller sizes)
        set  $choice(i) \leftarrow 1$ ;
    else (if node  $i$  has no alternatives left for a non-trivial coalition that would include  $i$ )
        set  $choice(i) \leftarrow 0$ ;
END DOALL
```

Step 4:

```
DOALL  $i = 1..n$ 
    send  $[V(i), C(i), choice(i), dec(i) = 0]$ 
END DOALL
```

Step 5:

```
DOALL  $i = 1..n$ 
    compare  $C(i)$  with  $C(j)$  received from one's neighbors;
    if there exists a clique  $\{i, j_1, j_2, \dots, j_l\}$  such that  $C(i) = C(j_1) = C(j_2) = \dots = C(j_l)$ 
    then set  $dec(i) \leftarrow 1$  (an agreement has been reached);
        broadcast group  $G = (i, j_1, j_2, \dots, j_l)$  and decision flag value  $dec(i) = 1$  to all neighbors
    else based on  $i$  and the priority (as defined by the relation  $\prec$  on the set of nodes  $1, 2, \dots, n$ )
        either DO NOTHING
        or change your mind:  $C(i) \leftarrow newchoice$  (from the list of candidate groups)
    END DOALL
END DO [* end of WHILE loop *]
```

ATSpace: A Middle Agent to Support Application Oriented Matchmaking and Brokering Services

Myeong-Wuk Jang, Amr Abdel Momen, Gul Agha

Department of Computer Science

University of Illinois at Urbana-Champaign, USA

{*mjang, amrmomen, agha*}@uiuc.edu

<http://osl.cs.uiuc.edu>

Abstract

An important problem for agents in open multiagent systems is how to find agents that match certain criteria. A number of middle agent services—such as matchmaking and brokering services—have been proposed to address this problem. However, the search capabilities of such services are relatively limited since the match criteria they use are relatively inflexible. We propose ATSpace, a model to support application-oriented matchmaking and brokering services. Application agents in ATSpace deliver their own search algorithms to a public tuple space which holds agent property data; the tuple space executes the search algorithms on this data. We show how the ATSpace model increases the dynamicity and flexibility of a middle agent service. Unfortunately, the model also introduces security threats: the data and access control restrictions in ATSpace may be compromised, and system availability may be affected. We describe some mechanisms to address these security threats.

1 Introduction

In multiagent systems, agents need to communicate with each other to accomplish their goals, and an important problem in open multiagent systems is the problem of finding other agents that match given criteria, called the *connection problem* [4]. When agents are designed or owned by the same organization, developers may be able to design agents which explicitly know the names of other agents that they need to communicate with. However in *open systems*, because an agent may be implemented by different groups, it is not feasible to let agents know the names of all agents that they may at some point need to communicate with.

Decker classifies middle agent services as either *matchmaking* (also called *Yellow Page*) services or *brokering* ser-

vices [5]. Matchmaking services (e.g. Directory Facilitator in FIPA platforms [7]) are passive services whose goal is to provide a client agent with a list of names of agents whose properties match its supplied criteria. The agent may later contact the matched agents to request services. On the other hand, brokering services (e.g. ActorSpace [1]) are active services that directly deliver a message (or a request) to the relevant agents on their client's behalf.

In both types of services, an agent advertises itself by sending a message which contains its name and a description of its characteristics to a *service manager*. A service manager may be implemented on top of a tuple space model such as Linda [3]; this involves imposing constraints on the format of the stored tuples and using Linda-supported primitives. Specifically, to implement matchmaking and brokering services on top of Linda, a tuple template may be used by the client agent to specify the matching criteria. However, the expressive power of a template is very limited; it consists of value constraints for its actual parameters and type constraints for its formal parameters. In order to overcome this limitation, Callsen's ActorSpace implementation used regular expressions in its search template [1]. Even though this implementation increased expressivity, its capability is still limited by the power of its regular expressions.

We propose *ATSpace*¹ (Active Tuple Spaces) to empower agents with the ability to provide arbitrary application-oriented search algorithms to the tuple space manager for execution on the tuple space. While ATSpace increases the dynamicity and flexibility of the tuple space model, it also introduces some security threats as codes developed by different groups with different interests are executed in the same space. We will discuss the implication of these threats and how they may be mitigated.

¹We will use *ATSpace* to refer the model for a middle agent to support application-oriented service, while we use an *atSpace* to refer an instance of ATSpace.

2 ATSpace

2.1 A Motivativig Example

We present a simple example to motivate the ATSpace model. Assume that a tuple space has information about seller agents (e.g., city and name) and the prices of the products they sell. A buyer agent wants to contact the two “best” seller agents who offer computers and whose location is within 50 miles of his city. A brokering service supplied by a generic tuple space implementation may not support the request of the buyer agent because, firstly, it may not support the “best two” primitive, and secondly, it may not store distance information between cities. The buyer agent is now opt to retrieve from the tuple space the complete tuples that are related to computer sellers, and then execute their own search algorithm on them. However, this approach entails the movement of large amount of data. In order to reduce communication overhead, ATSpace allows a sender agent to send its own search algorithm which may, for example, carry information about distances to the nearest cities.

2.2 Overall Architecture

ATSpace consists of three components: a tuple space, a message queue, and a tuple space manager (see Figure 1).

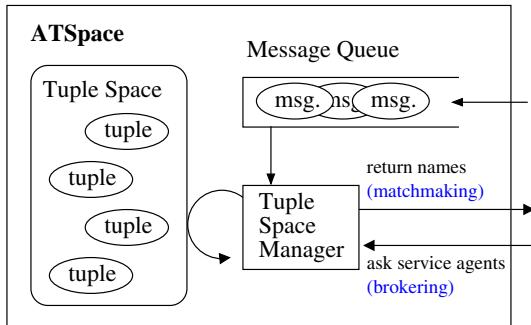


Figure 1. Basic Architecture of ATSpace

The tuple space is used as a shared pool for agent tuples, $\langle a, p_1, p_2, \dots, p_n \rangle$, which consists of a name field, a , and a property part, $P = p_1, p_2, \dots, p_n$ where $n \geq 1$; each tuple represents an agent whose name is given by the first field and whose characteristics are given by the subsequent fields. ATSpace enforces the rule that there cannot be more than one agent tuple whose agent names and property fields are identical at any time. However, an agent may register itself with different properties, and different agents may register themselves with the same property fields.

$$\forall t_i, t_j : i \neq j \rightarrow [(t_i.a = t_j.a) \rightarrow (t_i.p \neq t_j.p) \quad \&& \\ (t_i.p = t_j.p) \rightarrow (t_i.a \neq t_j.a)]$$

The message queue contains input messages that are received from other agents. Messages are classified into two types: *data input messages* and *service request messages*. A data input message includes a new agent tuple for insertion into the tuple space. A service request message includes either a tuple template or a mobile object. The template (or, alternately, the object) is used to search for agents with the appropriate agent tuples. A service message may optionally contain another field, called the *service call message field*, to facilitate the brokering service. A *mobile object* is an object that is provided by a service request agent; such objects have a pre-defined public method called `find`. The `find` method is called by the tuple space manager with tuples in this atSpace as a parameter; it returns names of agents selected by the search algorithm specified in the mobile object.

The tuple space manager retrieves names of service agents whose properties match a tuple template or which are selected by the mobile object. In case of a matchmaking service, it returns the names to the client agent. In case of brokering service, it forwards the service call message supplied by the client agent to the agents.

2.3 Operation Primitives

The ATSpace model supports the three general tuple space primitives: `write`, `read`, and `take`. In addition, ATSpace also provides primitives for the matchmaking and brokering services. The `searchOne` primitive is used to retrieve the name of a service agent that satisfies a given property, whereas the `searchAll` primitive is used to retrieve the names of all service agents that satisfy the property. The `deliverOne` primitive is used to forward a specified message to a service agent that matches the property, whereas the `deliverAll` is used to send this message to all such service agents. These matchmaking and brokering service primitives allow client agents to use mobile objects to support application-oriented search algorithm as well as a tuple template. `MobileObject` is used as an abstract class that defines the interface methods between a mobile object and the ATSpace. One of these methods is `find`, which may be used to provide the search algorithm to an atSpace.

When a client agent requires information about specific properties of service agents stored in an atSpace to make service call messages, the above matchmaking or brokering service primitives cannot be used. The `exec` primitive within a mobile object provides this service. The supplied mobile object has to implement the `doAction` method which when called by the atSpace with agent tuples, examines the properties of agents using the client agent application logic, creates different service call messages according to the properties, and then returns a list of agent messages

to the atSpace for delivery to the selected agents.

3 Security Issues

There are three important security problems in ATSpace.

Data Integrity A mobile object may not modify tuples owned by other agents.

Denial of Service A mobile object may not consume too much processing time or space of an atSpace and a client agent may not send repeatedly mobile objects to overload an atSpace.

Illegal Access A mobile object may not carry out unauthorized accesses or illegal operations.

We address the data integrity problem by blocking attempts to modify tuples. When a mobile object is executed by a tuple space manager, the manager makes a copy of tuples and then sends the copy to the `find` or `doAction` method of the mobile object. Therefore, even when a malicious agent changes some tuples, the original tuples are not affected by the modification. However, when the number of tuples in the tuple space is very large, this solution requires extra memory and computational resources. For better performance, the creator of an atSpace may select the option to deliver a shallow copy of the original tuples to mobile objects instead of a deep copy, although this will violate the integrity of tuples if an agent tries to delete or change its tuple. We are currently investigating under what conditions a use of a shallow copy may be sufficient.

To address denial of service by consuming all processor cycles, we deploy user-level thread scheduling. Figure 2 depicts the extension of the tuple space manager to achieve this. When a mobile object arrives, the object is executed as a thread, and its priority is set to high. If the thread executes for a long time, its priority is continually downgraded. Moreover, if the running time of a mobile object exceeds a certain limit, it may be destroyed by the Tuple Space Manager; in this case, a message is sent to its supplier informing it of the destruction. To incorporate these restrictions, we have extended the architecture of ATSpace by implementing job queues.

To prevent unauthorized accesses, if an atSpace is created with an *access key*, then this key must accompany every message sent from service requester agents. In this case, agents are allowed to modify only their own tuples. This prevents removal or modification of tuples by unauthorized agents.

4 Evaluation

Figure 3 shows the advantage of ATSpace compared to a matchmaking service which provides the same semantic

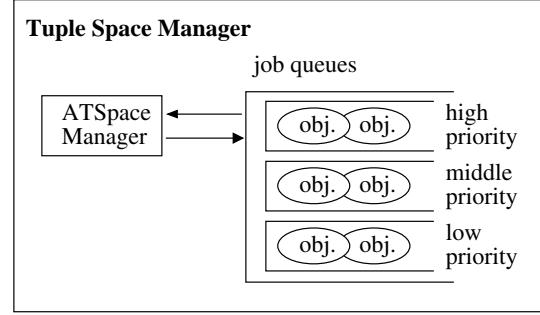


Figure 2. Extended Architecture of ATSpace

in UAV simulation (see [9] for details of this simulation). In these experiments, the UAVs use either an active brokering service or a matchmaking service to find their neighboring UAVs. In both cases, the middle agent includes information about the locations of UAVs. In case of the active brokering service, UAVs send mobile objects to the middle agent while UAVs using matchmaking service send tuple templates. The simulation time for each run is around 35 minutes.

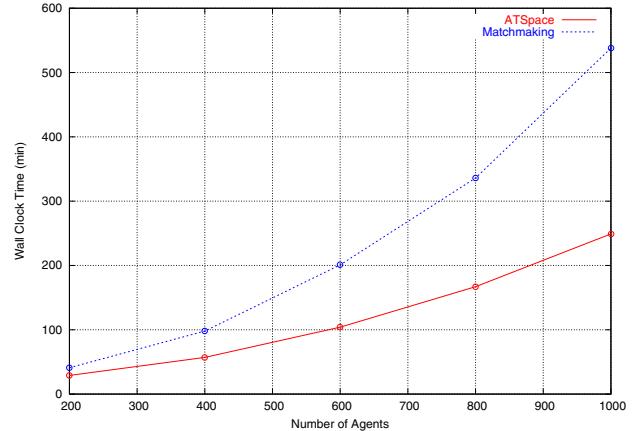


Figure 3. Wall Clock Time for ATSpace and Matchmaking Services

When the number of agents is small, the difference between the two approaches is not significant. However, as the number of agents is increased, the difference becomes significant.

5 Related Work

Our work is related to *Linda* [3] and its variations, such as *JavaSpaces* and *TSpaces* [10, 13]. In these models, processes communicate with other processes through a shared common space called a *tuple space* without considering ref-

erences or names of other processes. From the middle agent perspective, *Directory Facilitator* in the *FIPA platform* and *Broker Agent* in *InfoSleuth* are related to our research [7, 8]. However, these systems do not support customizable matching algorithm.

Some work has been done to extend the matching capability in the tuple space model. *Berlinda* allows a concrete entry class to extend the matching function [14]. However, this approach does not allow the matching function to be changed during execution time. *OpenSpaces* provides a mechanism to change matching polices during the execution time [6]. *OpenSpaces* groups entries in its space into classes and allows each class to have its individual matching algorithm. A manager for each class of entries can change the matching algorithm during execution time. All agents that use entries under a given class are affected by any change to its matching algorithm. This is in contrast to ATSpace where each agent can supply its own matching algorithm without affecting other agents. Another difference between OpenSpaces and ATSpace is that the former requires a registration step before putting the new matching algorithm into action.

TuCSOn and *MARS* provide programmable coordination mechanisms for agents through Linda-like tuple spaces to extend the expressive power of tuple spaces [2, 11]. However, they differ in the way they approach the expressiveness problem; while *TuCSOn* and *MARS* use reactive tuples to extend the expressive power of tuple spaces, ATSpace uses mobile objects to support search algorithms defined by client agents. A reactive tuple handles a certain type of tuples and affects various clients, whereas a mobile object handles various types of tuples and affects only its creator agent. Also, these approaches do not provide an execution environment for client agents. Therefore, these may be considered as orthogonal approaches and can be combined with our approach together.

6 Conclusion

ATSpace works as a common shared space to exchange data among agents, a middle agent to support matchmaking and brokering services, and an execution environment for mobile objects utilizing data on its space. Our experiments with a UAV surveillance task show that the model may be effective in reducing coordination costs. We described some security threats that arise when using mobile objects for agent coordination, along with some mechanisms we use to mitigate them. We are currently incorporating memory use restrictions into the architecture and considering mechanisms to address denial of service attacks that may be caused by flooding the network [12].

Acknowledgements

This research is sponsored by the DARPA ITO under contract number F30602-00-2-0586.

References

- [1] G. Agha and C. Callsen. ActorSpaces: An Open Distributed Programming Paradigm. In *Proceedings of the 4th ACM Symposium on Principles & Practice of Parallel Programming*, pages 23–32, May 1993.
- [2] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: a Programmable Coordination Architecture for Mobile Agents. *IEEE Computing*, 4(4):26–35, 2000.
- [3] N. Carreiro and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.
- [4] R. Davis and R. Smith. Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence*, 20(1):63–109, January 1983.
- [5] K. Decker, M. Williamann, and K. Sycara. Matchmaking and Brokering. In *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS-96)*, Kyoto, Japan, December 1996.
- [6] S. Ducasse, T. Hofmann, and O. Nierstrasz. OpenSpaces: An Object-Oriented Framework for Reconfigurable Coordination Spaces. In A. Porto and G. Roman, editors, *Coordination Languages and Models, LNCS 1906*, pages 1–19, Limassol, Cyprus, September 2000.
- [7] Foundation for Intelligent Physical Agents. *SC00023J: FIPA Agent Management Specification*, December 2002. <http://www.fipa.org/specs/fipa00023/>.
- [8] N. Jacobs and R. Shea. The Role of Java in InfoSleuth: Agent-based Exploitation of Heterogeneous Information Resources. In *Proceedings of Intranet-96 Java Developers Conference*, April 1996.
- [9] M. Jang, A. A. Momen, and G. Agha. A Flexible Coordination Framework for Application-Oriented Matchmaking and Brokering Services. Technical Report UIUCDCS-R-2004-2430, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004.
- [10] T. Lehman, S. McLaughry, and P. Wyckoff. TSpaces: The Next Wave. In *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS-32)*, January 1999.
- [11] A. Omicini and F. Zambonelli. TuCSOn: a Coordination Model for Mobile Information Agents. In *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, Pisa, Italy, June 1998.
- [12] C. Shields. What do we mean by Network Denial of Service? In *Proceedings of the 2002 IEEE Workshop on Information Assurance and Security*, pages 17–19, United States Military Academy, West Point, NY, June 2002.
- [13] Sun Microsystems. *JavaSpacesTM Service Specification*, ver. 2.0, June 2003. <http://java.sun.com/products/jini/specs>.
- [14] R. Tolksdorf. Berlinda: An Object-oriented Platform for Implementing Coordination Language in Java. In *Proceedings of COORDINATION '97 (Coordination Languages and Models), LNCS 1282*, pages 430–433. Springer-Verlag, 1997.

Learning Continuous Time Markov Chains from Sample Executions

Koushik Sen, Mahesh Viswanathan, Gul Agha

Department of Computer Science

University of Illinois at Urbana Champaign

{ksen, vmahesh, agha}@cs.uiuc.edu

Abstract

Continuous-time Markov Chains (CTMCs) are an important class of stochastic models that have been used to model and analyze a variety of practical systems. In this paper we present an algorithm to learn and synthesize a CTMC model from sample executions of a system. Apart from its theoretical interest, we expect our algorithm to be useful in verifying black-box probabilistic systems and in compositionally verifying stochastic components interacting with unknown environments. We have implemented the algorithm and found it to be effective in learning CTMCs underlying practical systems from sample runs.

1. Introduction

Stochastic models such as continuous-time Markov chains (CTMCs) [22] are widely used to model practical software systems and analyze their performance and reliability. Before building a complex software system, CTMCs are generated from higher-level specifications, like queueing networks, stochastic process algebra [14, 11], or stochastic Petri-nets [4]. These models are then used for quantitative evaluation of reliability and performance for example to determine the throughput of production lines, to calculate average failure time of systems, or to find other reliability or performance bottlenecks of the system. Once a model has been validated against performance and reliability requirements, the system is implemented. However, even if a model has been carefully validated, the implementation may not conform to the model. There are two potential sources of error: first, there could be bugs introduced when translating the design into system code, and second the estimated values of various parameters used in constructing the stochastic model may differ considerably from the actual values in the deployed system. To catch such potential post-implementation problems, testing for performance and reliability is performed by running the system a large number of times in the real environment and checking for reliability problems or performance bottlenecks. However, because it is difficult achieve complete *coverage* during testing, despite its evident importance in practice, testing fails to guarantee the full correctness of a deployed system.

An approach that tries to leverage the benefits of formal analysis - usually done in the design phase - to the post-implementation phase, is *learning* the model from sample executions of the system and then formally verifying the learnt model against the design specification. This approach has been fruitfully used to model-check unknown, *black-box* systems [9] and to learn unknown environments to assist in compositional verification of systems [7]. Both these efforts apply to non-deterministic, discrete systems and have not been extended to more general stochastic systems. While there are several machine learning algorithms on grammatical inference [6, 1, 8, 19, 5] that have been successfully applied to pattern recognition, speech recognition, natural language processing and several other domains, there are no algorithms in the literature that can learn the real-time, stochastic models that are typically used to model systems in formal verification. In this paper, we address this problem by presenting an algorithm that given execution traces (possibly obtained by running the deployed system during testing) of the system, infers a CTMC model that could have generated the traces according to the observed distribution. The learned CTMC can then be used by existing probabilistic model-checking [17, 12, 21, 24] and performance evaluation tools [14, 4] for further analysis and thereby helping to find bugs in post-implementation phase. The learning algorithm may also potentially be used to perform automatic compositional verification (as in [7]) for stochastic systems. A closely related work is given in [23] where they learn continuous-time hidden Markov models to do performance evaluation. However, they fix the size of the continuous-time hidden Markov model before learning. This can be restrictive if the system cannot be modelled by a continuous-time hidden Markov model of given size. In our approach there is no such restriction.

We present an algorithm and show that it correctly identifies the CTMC model in the limit [8] when it is given samples drawn from a distribution generated by a CTMC. One technical difficulty when talking about samples drawn from a CTMC is that traditionally CTMCs are unlabeled, and so they only have runs, which are sequences of states that are traversed and not traces. However, the problem is that when samples are drawn from an implementation get-

ting information that uniquely identifies states is expensive and impractical, and can lead to the construction of a very large model which does not collapse equivalent states. To address this difficulty, we introduce the model of an *Edge Labeled Continuous-time Markov Chain* ($CTMC_L$) where edges are labeled from a finite set of alphabet and traces are sequences of edge labels which are given to the learning algorithm.

Our algorithm is based on the state merging paradigm introduced and used in RPNI [19] and ALERGIA [6]. The samples provided to the learning algorithm are used to construct what we call a *prefix-tree continuous-time Markov chain*. Such Markov chains are the simplest CTMC that are consistent with the samples. The algorithm then progressively generalizes this model (i.e., produces models with additional behaviors) by merging pairs of states about whose “equivalence” the sample has evidence. Since the traces do not have complete state information about the original CTMC states, statistical information present in the samples is used to distinguish states. Our key algorithmic insight is in determining the statistical tests that can be used to conclude the equivalence of states with a given confidence. The candidate states that are tested for equivalence by the algorithm are done in a carefully chosen order (as in RPNI and ALERGIA) to ensure that the algorithm runs in time polynomial in sample size. The algorithm terminates when it has tested for all possible merges. Like all algorithms that learn in the limit, we show that this algorithm learns the correct CTMC given a sufficiently large sample. Our proof that the algorithm learns in the limit relies on a novel method to bound the error probability of our statistical tests. The CTMC that the algorithm learns may be much smaller than the implementation, since it merges all potentially equivalent states, and it only generates the *reachable* portion of the implementation. This can be particularly beneficial in the context of formal verification; the running time and space requirements of verification algorithms depend on the size of the reachable portion of the model. We have implemented our algorithm in Java and experimented by learning some example systems encountered in practice.

The rest of the paper is organized as follows. We give the preliminary definitions and notations in Section 2, followed by the learning algorithm in Section 3. In Section 4 we prove that the learned CTMC converges to the original CTMC in the limit. We report our initial experimental results in Section 5 and conclude in Section 6.

2. Preliminaries

We recall some basic concepts related to CTMCs. Our presentation of this material is slightly non-standard in that we consider CTMCs that have labels both on the edges and the states. In what follows we assume AP to be a finite set of atomic propositions that are used in describing reliability and performance constraints.

Definition 1 An Edge Labeled Continuous-time Markov Chain ($CTMC_L$) is a tuple $\mathcal{M} = (S, \Sigma, s_0, \delta, \rho, L)$ where

- S is a finite set of states,
- Σ is a finite alphabet of edge labels,
- $s_0 \in S$ is the initial state,
- $\delta: S \times \Sigma \rightarrow S$ is a partial function which maps a state and an alphabet to the next state,
- $\rho: S \times \Sigma \rightarrow \mathbb{R}_{\geq 0}$ is a function which returns a positive real, called rate, associated with the transition. We assume that $\rho(s, a) = 0$ if and only if $\delta(s, a)$ is not defined.
- $L: S \rightarrow 2^{AP}$ is a function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in s .

A $CTMC_L$ defined as above is deterministic in the sense that for a given state $s \in S$ and an alphabet $a \in \Sigma$ the state reached from s by the edge labeled a is unique if it exists. Intuitively, the probability of moving from state s to state s' along the edge labeled a within time t is given by $(1 - e^{-\rho(s,a)t})$. This probability corresponds to the cumulative probability of an exponential distribution with rate $\rho(s, a)$. For a given state s , if there are more than one alphabet $a \in \Sigma$ such that $\rho(s, a) > 0$ then there is a competition between the transitions. More precisely, for each transition $s \rightarrow \delta(s, a)$ from s for which $\rho(s, a) > 0$, a random time t is sampled from the exponential distribution with rate $\rho(s, a)$. Then the transition corresponding to the lowest sampled time is taken. The probability to move from a state s to another state, along the edge a , within t time units i.e. the time sampled for the transition corresponding to $s \rightarrow \delta(s, a)$ is minimum, is given by

$$\mathbf{P}(s, a, t) = \frac{\rho(s, a)}{\mathbf{E}(s)} \left(1 - e^{-\mathbf{E}(s)t}\right)$$

where $\mathbf{E}(s) = \sum_{a \in \Sigma} \rho(s, a)$ is the total rate at which any transition from the state s is taken. In other words, the probability of leaving the state s within t time units is $(1 - e^{-\mathbf{E}(s)t})$. This is because the distribution for the minimum time among all edges is exponential with rate $\mathbf{E}(s)$. Thus we can see the probability of moving from a state s along the edge a is the probability of staying at the state s for less than t time units times the probability of taking the edge a . The probability of taking the edge a from state s is thus given by

$$\mathbf{P}(s, a) = \frac{\rho(s, a)}{\mathbf{E}(s)}$$

When $\mathbf{E}(s) = 0$, we define $\mathbf{P}(s, a) = 0$ for every a .

Paths and Probability Space A path τ starting at state s is a finite or infinite sequence $l_0 \xrightarrow{(a_1, t_1)} l_1 \xrightarrow{(a_2, t_2)} l_2 \xrightarrow{(a_3, t_3)} \dots$ such that there is a corresponding sequence $v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} v_2 \xrightarrow{a_3} \dots$ of states with $s = v_0$, $L(v_i) = l_i$, $t_i \in \mathbb{R}_{\geq 0}$, $\delta(v_i, a_{i+1}) = v_{i+1}$ for all i . For a path τ from s , $\tau[s, i] = v_i$ is the i^{th} state of the path and $\eta[\tau, s, i] = t_i$ is the time spent in state v_{i-1} . A *maximal* path starting at state s is a path τ starting at s such that it is either infinite or (if finite) $E(\tau[s, n]) = 0$, where n is the length of τ . The set of all maximal paths starting at state s is denoted by $\text{Path}(s)$; the set of all (maximal) paths in a CTMC \mathcal{M} , denoted by $\text{Path}(\mathcal{M})$, is taken to be $\text{Path}(s_0)$, where s_0 is the initial state.

Let $\pi = l_0 \xrightarrow{a_1} l_1 \xrightarrow{a_2} \dots \xrightarrow{a_k} l_k$ be a finite sequence such that there is a sequence of states s_0, s_1, \dots, s_k such that s_0 is the initial state, $L(s_i) = l_i$ and $\delta(s_i, a_{i+1}) = s_{i+1}$; $\pi[i]$ is used to denote state s_i in the sequence corresponding to π . Let I_1, I_2, \dots, I_k be non-empty intervals in $\mathbb{R}_{\geq 0}$. Then $C(l_0, (a_1, I_1), l_1, \dots, (a_k, I_k), l_k)$ denotes a *cylinder set* consisting of all paths $\tau \in \text{Path}(s_0)$ such that $\tau[s_0, i] = \pi[i]$ (for $i \leq k$) and $\eta[\tau, s_0, i] \in I_i$. Let \mathcal{B} be the smallest σ -algebra on $\text{Path}(s_0)$ which contains all the cylinders $C(l_0, (a_1, I_1), l_1, \dots, (a_k, I_k), l_k)$. The probability measure over \mathcal{B} is the unique measure inductively defined as $\text{Path}(C(l_0)) = 1$, if $L(s_0) = l_0$ and for $k > 0$ as

$$\begin{aligned} \text{Prob}(C(l_0, (a_1, I_1), l_1, \dots, l_{k-1}, (a_k, I_k), l_k)) \\ = \text{Prob}(C(l_0, (a_1, I_1), l_1, \dots, l_{k-1})) \cdot \\ \mathbf{P}(s_{k-1}, a_k) \cdot (e^{-\mathbf{E}(s_{k-1})\ell} - e^{-\mathbf{E}(s_{k-1})u}) \end{aligned}$$

where $\ell = \inf I_k$ and $u = \sup I_k$, and $s_{k-1} = \pi[k-1]$

3. Learning Edge Labeled CTMCs

The learning problem considered in this paper falls under the category of stochastic grammatical inference [6, 16, 20, 5]. In stochastic grammatical inference, samples are taken from a stochastic language. Given these samples the stochastic language is learned by finding statistical regularity among the samples. The parameters for the different distributions determining the language are also estimated from the relative frequencies of the samples. For most of these learning algorithms it has been shown that they can learn the stochastic language in the *limit* i.e. if the number of samples tends towards infinity then the learned language is the same as the language that generated the sample. All these algorithms essentially follow the same technique: they build a prefix-tree automata which stores exactly the same samples and then they test and merge possibly equivalent states.

We present the algorithm for learning edge labeled CTMCs. We first consider the issue of how to generate and reason about behaviors (visible execution traces) of finite length, given that traditionally the behaviors are assumed to be of infinite length. We then present some concepts that are used in the algorithm. After this we present

the algorithm, whose proof of correctness appears in Section 4.

3.1. Generating Samples

In this paper we consider the problem of learning CTMC_L from examples generated by simulating a system under investigation. The way CTMC_L is formally defined in Section 2, all behaviors are *maximal* executions, and maximal executions are typically infinite. This creates a technical difficulty namely what the samples appropriate for learning are. To overcome this problem we define a finitary version of CTMC_L called *Finitary Edge Labeled Continuous-time Markov Chains* (CTMC_L^f) which is a CTMC_L , with a non-zero stopping probability in any state. This allows one to generate and reason about behaviors of finite length. It is important to note however, that use of CTMC_L^f is merely a technical tool. Our primary goal is to learn the underlying CTMC_L and as we shall see in Proposition 3, we can achieve this by learning the CTMC_L^f . Moreover in this effort, the specific value of the stopping probability that we use does not influence the correctness of the result. We present the formal definition of a CTMC_L^f .

Definition 2 A Finitary Edge Labeled Continuous-time Markov Chain (CTMC_L^f) is a pair $\mathcal{F} = (\mathcal{M}, p)$ where \mathcal{M} is a CTMC_L and p denotes the stopping probability in any state s of \mathcal{M}

There exists a trivial surjection $\Theta: (\mathcal{M}, p) \mapsto \mathcal{M}$.

A finite sequence $\tau = l_0 \xrightarrow{(a_1, t_1)} l_1 \xrightarrow{(a_2, t_2)} l_2 \dots \xrightarrow{(a_n, t_n)} l_n$ is a path of the CTMC_L^f $\mathcal{F} = (\mathcal{M}, p)$ starting from a state s iff it is path (not necessarily maximal) of \mathcal{M} starting from s . The set of paths starting from state s is denoted by $\text{Path}(s)$. The i^{th} state of path τ from s and the time spent in the i^{th} state are defined similarly, and are denoted by $\tau[s, i]$ and $\eta[\tau, s, i]$, respectively. The σ -field corresponding to a CTMC_L^f is defined analogously to that of a CTMC_L . For the path τ from state s , the probability that the CTMC_L^f exhibits such a path is given by

$$\begin{aligned} \text{Prob}_{\mathcal{F}}(\tau, s) = \\ (1-p) \cdot \mathbf{P}(\tau[s, 0], a_1, t_1) \cdot (1-p) \cdot \mathbf{P}(\tau[s, 1], a_2, t_2) \\ \dots (1-p) \cdot \mathbf{P}(\tau[s, n-1], a_n, t_n) \cdot p \end{aligned}$$

Given a CTMC_L we extend it to a CTMC_L^f by associating a known probability p_{\perp} (say $p_{\perp} = 0.1$) as the stopping probability. The CTMC_L^f thus obtained is then simulated to get a multi-set of finite samples which we treat as the multi-set of examples for learning. In our algorithm we will assume that we are given a finite multi-set of examples from a CTMC_L \mathcal{M} extended with a known stopping probability p to a CTMC_L^f . Our goal will be to learn \mathcal{M} from the multi-set of examples.

Note that for a given implemented system, which can be seen as a software program, an example can be generated in the following way. Let s_0 be the initial state of the program. Then add $l_0 = L(s_0)$ to the example sequence. We set a probability $p_0 = 0.1$. With probability p_0 return the current sequence as an example. With probability $1 - p_0$ execute the next instruction of the program. If the execution of the instruction a_i takes time t_i and results in the change of state from s_{i-1} to s_i then add $\xrightarrow{(a_i, t_i)} L(s_i)$ to the example sequence.

3.2. Definitions

We next define the notations and the concepts that we will use to describe the learning algorithm. Given a $CTMC_L \mathcal{M} = (S, \Sigma, s_0, \delta, \rho, L)$ we can extend the definition of δ as follows:

$$\delta(s, \lambda) = s \text{ where } \lambda \text{ is the empty string}$$

$$\delta(s, xa) = \delta(\delta(s, x), a) \text{ where } x \in \Sigma^* \text{ and } a \in \Sigma$$

$$\delta(s, a) = \perp \text{ if } \delta(s, a) \text{ is not defined}$$

$$\delta(s, xa) = \perp \text{ if } \delta(s, x) = \perp \text{ or } \delta(\delta(s, x), a) \text{ is undefined}$$

For a given example $\tau = l_0 \xrightarrow{(a_1, t_1)} l_1 \xrightarrow{(a_2, t_2)} l_2 \dots \xrightarrow{(a_n, t_n)} l_n$, let $\tau|_\Sigma$ be the string $a_1 a_2 \dots a_n$. We use $Pr(\tau)$ to denote the set $\{x \mid \exists y: xy = \tau|_\Sigma\}$, that is, $Pr(\tau)$ is the set of all prefixes of $\tau|_\Sigma$. Given a multi-set I^+ of examples, let $Pr(I^+)$ be the set $\bigcup_{\tau \in I^+} Pr(\tau)$. If there exists an example $l_0 \xrightarrow{(a_1, t_1)} l_1 \xrightarrow{(a_2, t_2)} l_2 \dots \xrightarrow{(a_i, t_i)} l_i \dots \xrightarrow{(a_n, t_n)} l_n$ in I^+ such that $x = a_1 a_2 \dots a_i$ then we define $L(x, I^+) = l_i$.

Let $n(x, I^+)$ be the number of $\tau \in I^+$ such that $x \in Pr(\tau)$ and let $n'(x, I^+)$ be $n(x, I^+)$ minus the number of $x \in I^+$. Thus $n(x, I^+)$ counts the number of examples in I^+ for which x is a prefix and $n'(x, I^+)$ is the number of examples in I^+ for which x is prefix and length of x is less than the length of the example. For $\tau = l_0 \xrightarrow{(a_1, t_1)} l_1 \xrightarrow{(a_2, t_2)} l_2 \dots \xrightarrow{(a_i, t_i)} l_i \dots \xrightarrow{(a_n, t_n)} l_n$, if $x = a_1 a_2 \dots a_{i-1}$ and $a = a_i$, then $\theta(x, a, \tau) = t_i$ and 0 otherwise; in other words, $\theta(x, a, \tau)$ denotes the time spent in the state reached after x in τ . We define $\hat{\theta}(x, I^+)$ and $\hat{p}(x, a, I^+)$ as follows:

$$\begin{aligned} \hat{\theta}(x, I^+) &= \begin{cases} \frac{\sum_{a \in \Sigma} \sum_{\tau \in I^+} \theta(x, a, \tau)}{n'(x, I^+)} & \text{if } n'(x, I^+) > 0 \\ 0 & \text{otherwise} \end{cases} \\ \hat{p}(x, a, I^+) &= \begin{cases} \frac{n(xa, I^+)}{n'(x, I^+)} & \text{if } n'(x, I^+) > 0 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Note that $\hat{\theta}(x, I^+)$ gives an estimate of $1/\mathbf{E}(s)$ where s is the state $\delta(s_0, x)$ and $\hat{p}(x, a, I^+)$ gives an estimate of $\mathbf{P}(s, a)$.

Given a multi-set I^+ of examples we first construct a prefix-tree $CTMC_L^f$ defined as follows.

Definition 3 The prefix-tree $CTMC_L^f$ for a multi-set of examples I^+ is a $CTMC_L^f$ $PCTMC(I^+) = ((S, \Sigma, s_0, \delta, \rho, L), p)$, where

$$1. S = Pr(I^+)$$

$$2. s_0 = \lambda \text{ (the empty string)}$$

$$3. \delta(x, a) = \begin{cases} xa & \text{if } xa \in S \\ \perp & \text{otherwise} \end{cases}$$

$$4. \mathbf{E}(x) = 1/\hat{\theta}(x, I^+)$$

$$5. \mathbf{P}(x, a) = \hat{p}(x, a, I^+)$$

$$6. \rho(x, a) = \mathbf{P}(x, a)\mathbf{E}(x)$$

$$7. L(x) = L(x, I^+)$$

$$8. p \text{ is the stopping probability associated with the } CTMC_L^f \text{ that generated the examples.}$$

A $PCTMC(I^+)$ is an $CTMC_L^f$ consistent with the examples in I^+ in the sense that for every example in I^+ there is a corresponding path in the $CTMC_L^f$.

The learning algorithm proceeds by generalizing the initial guess, $PCTMC(I^+)$, by merging *equivalent* states. The formal definition of when two states are equivalent is now presented.

Definition 4 Given a $CTMC_L \mathcal{M} = (S, \Sigma, s_0, \delta, \rho, L)$, a relation $R \subseteq S \times S$ is said to be stable relation if and only if for any $s, s' \in S$ such that $(s, s') \in R$, we have

$$a) L(s) = L(s')$$

$$b) \mathbf{E}(s) = \mathbf{E}(s')$$

$$c) \text{for all } a \in \Sigma \text{ if there exists } t \in S \text{ such that } \delta(s, a) = t \text{ then there exists a } t' \in S \text{ such that } \delta(s', a) = t', \mathbf{P}(s, a) = \mathbf{P}(s', a) \text{ and } (t, t') \in R, \text{ and conversely}$$

$$d) \text{for all } a \in \Sigma \text{ if there exists } t' \in S \text{ such that } \delta(s', a) = t' \text{ then there exists a } t \in S \text{ such that } \delta(s, a) = t, \mathbf{P}(s', a) = \mathbf{P}(s, a) \text{ and } (t', t) \in R.$$

Two states s and s' in $CTMC_L \mathcal{M}$ are said to be equivalent ($s \equiv s'$) if and only if there is a stable relation R such that $(s, s') \in R$.

The correctness of learning algorithm crucially depends on the fact that merging two equivalent states results in a $CTMC_L$ that generates the same distribution. But before we state and prove this formally we make a simple observation about equivalent states.

Lemma 1 Let $\mathcal{F} = (\mathcal{M}, p)$ be an $CTMC_L^f$ and $s \equiv s'$. τ is a path starting from s iff τ is a path starting from s' . Moreover $Prob_{\mathcal{F}}(\tau, s) = Prob_{\mathcal{F}}(\tau, s')$.

Proof: Let $\tau = l_0 \xrightarrow{(a_1, t_1)} l_1 \xrightarrow{(a_2, t_2)} l_2 \dots \xrightarrow{(a_n, t_n)} l_n$ be a path starting from s . There is a sequence $v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} \dots v_n$ of states such that $v_0 = s$. Since $s \equiv s'$ and $\delta(s, a_1) = v_1$, there must be a state u_1 such that $\delta(s', a_1) = u_1$ and $u_1 \equiv v_1$. Continuing inductively, we can construct a sequence of states $u_0 \xrightarrow{a_1} u_1 \xrightarrow{a_2} \dots u_n$, such that $u_0 = s'$

and $u_i \equiv v_i$. Hence τ is also a path starting from s' . Furthermore, since $u_i \equiv v_i$, we know that $\mathbf{E}(u_i) = \mathbf{E}(v_i)$, and $\mathbf{P}(u_i, a_{i+1}) = \mathbf{P}(v_i, a_{i+1})$ and hence $\text{Prob}_{\mathcal{F}}(\tau, s) = \text{Prob}_{\mathcal{F}}(\tau, s')$. For paths starting from s' , the argument is symmetric.

Definition 5 Two CTMC_L^f 's \mathcal{F} and \mathcal{F}' with initial states s_0 and s'_0 , respectively, are said to be equivalent if $\text{Path}(\mathcal{F}) = \text{Path}(\mathcal{F}')$ and for every $\tau \in \text{Path}(\mathcal{F})$, $\text{Prob}_{\mathcal{F}}(\tau, s_0) = \text{Prob}_{\mathcal{F}'}(\tau, s'_0)$.

For a CTMC_L $\mathcal{M} = (S, \Sigma, s_0, \delta, \rho, L)$, the *minimal* CTMC_L is defined to be the quotient of \mathcal{M} with respect to the equivalence relation on states. Formally, the minimal CTMC_L is $\mathcal{M}' = (S', \Sigma, s'_0, \delta', \rho', L')$ such that

1. S' are the equivalence classes of S with respect to \equiv ,
2. $s'_0 = [s_0]$, the equivalence class of s_0
3. $\delta'([s], a) = [s']$ iff $\delta(s, a) = s'$
4. $\rho'([s], a) = \rho(s, a)$ and
5. $L'([s]) = L(s)$

Observe, that this is well-defined, because of the way \equiv is defined.

Proposition 2 Let $\mathcal{F} = (\mathcal{M}, p)$ be a CTMC_L^f . Then $\mathcal{F}' = (\mathcal{M}', p)$ is equivalent to \mathcal{F} where \mathcal{M}' is the minimal CTMC_L corresponding to \mathcal{M} .

Proof: The proof is a straightforward consequence of the definition of the minimal CTMC_L \mathcal{M}' . It relies on the observation that for a path τ , $v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} \dots v_n$ is a sequence of states visited in \mathcal{M} iff $[v_0] \xrightarrow{a_1} [v_1] \xrightarrow{a_2} \dots [v_n]$ in \mathcal{M}' . Furthermore, since $\rho'([s], a) = \rho(s, a)$, the probabilities are also the same.

We conclude the section with the observation that for equivalent CTMC_L^f 's with the same stopping probability, the associated CTMC_L 's define the same probability space on the set of paths. This next proposition together with Proposition 2 shows that given any CTMC_L , we can always construct a smaller equivalent CTMC_L by merging equivalent states, thus providing mathematical justification for our algorithm.

Proposition 3 Let $\mathcal{F} = (\mathcal{M}, p)$ and $\mathcal{F}' = (\mathcal{M}', p)$ be two CTMC_L^f 's with the same stopping probability p . Then the probability spaces defined by \mathcal{M} and \mathcal{M}' are the same.

We skip the proof of this proposition in the interests of space. However, we would like to point out some of the important consequences of Proposition 3. First is that if we learn an CTMC_L^f that has the same stopping probability as the one that was used to generate the samples from the system, then the underlying CTMC_L 's are also equivalent in terms of the distribution on traces they generate. Second, the specific value of the stopping probability plays no role

in proving the correctness of our learning algorithm. It may have an effect in terms of the length of traces produced and the number of traces needed to learn. The right choice of the stopping probability is thus one that is determined by the empirical constraints that one is working in.

3.3. Learning Algorithm

```

algorithm learnCTMC
Input:  $I^+$  : a multi-set of examples
         $\alpha$  : confidence level
Output:  $\text{CTMC}_L$ 
begin
     $A \leftarrow \text{PCTMC}(I^+)$ 
    for  $i = 2$  to  $|A|$  do
        for  $j = 1$  to  $i - 1$  do
            if compatible( $s_i, s_j, \alpha, I^+$ ) then
                 $A \leftarrow \text{merge}(A, s_i, s_j)$ 
                 $A \leftarrow \text{determinize}(A)$ 
                exit  $j$ -loop
            endif
    return  $A$ 
end

```

Figure 1. Algorithm to learn CTMC_L^f

The algorithm for CTMC_L learning, described in Figure 1, first constructs the prefix-tree $\text{CTMC}_L^f A = \text{PCTMC}(I^+)$ from the multi-set of examples I^+ . We assume that the states in A are ordered in lexicographic order.¹ Let $|A|$ be the number of states in A . The algorithm then tries to merge pairs of states in A that are equivalent in a quadratic loop, i.e. for all i from 1 to $|A|$ the algorithm tries to merge s_i with the states s_1, s_2, \dots, s_{i-1} in that order. If two states s_i and s_j are equivalent they are merged using the method $\text{merge}(A, s_i, s_j)$. The smallest state in a block of merged states is used to represent the whole block. After every merge of state s_i and s_j the resulting CTMC_L^f may be non-deterministic. However, equivalence of s_i and s_j implies that each successor of s_i is equivalent to the corresponding successor of s_j . This means that those successors should also get merged. To ensure this the method $\text{determinize}(A)$ described in Figure 2 is invoked which removes the non-determinism in A by a sequence of merges. After every merge the probabilities $\mathbf{P}(s, a)$ and the rates $\mathbf{E}(s)$ are re-computed for every state as there is more information available at every state. The algorithm stops when no more merging is possible.

¹ For $\Sigma = \{a, b\}$, the lexicographic ordering is $\lambda, a, b, aa, ab, ba, bb, aaa, \dots$

```

algorithm determinize
Input: A
Output: CTMCL
begin
  while( $\exists s, a \in A : s', s'' \in \delta(s, a)$ ) do
    A  $\leftarrow$  merge(A, s', s'')
  return A
end

```

Figure 2. *determinize* removes non-determinism

Now the observations Proposition 2 and 3 together suggest that the above algorithm would be correct if indeed we could test for equivalence of two states. This, however, is not the case, as A is built from experimental data. However, we approximately check the equivalence of two states recursively through statistical hypothesis testing [15, 18]. We say that two states s_i and s_j are compatible, denoted by $s_i \approx s_j$, if $L(s) = L(s')$, $\mathbf{E}(s) \sim \mathbf{E}(s')$, for all $a \in \Sigma$, $\mathbf{P}(s_i, a) \sim \mathbf{P}(s_j, a)$, and $\delta(s_i, a) \approx \delta(s_j, a)$, where $\mathbf{E}(s) \sim \mathbf{E}(s')$ means that $\mathbf{E}(s)$ and $\mathbf{E}(s')$ are equal within some statistical uncertainty and similarly for $\mathbf{P}(s_i, a) \sim \mathbf{P}(s_j, a)$. The decision for compatibility is made using the function *compatible* described in Figure 3.

```

algorithm compatible
Input: x, y, I+,  $\alpha$ 
Output: boolean
begin
  if  $L(x, I^+) \neq L(y, I^+)$  then
    return FALSE
  if differentExpMeans( $\hat{\theta}(x, I^+), n'(x, I^+)$ ,
                       $\hat{\theta}(y, I^+), n'(y, I^+), \alpha$ ) then
    return FALSE
  for  $\forall a \in \Sigma$ 
    if differentBerMeans( $\hat{p}(x, a, I^+), n(xa, I^+)$ ,
                         $\hat{p}(y, a, I^+), n(ya, I^+), \alpha$ ) then
      return FALSE
  if not compatible( $\delta(x, a), \delta(y, a), I^+, \alpha$ ) then
    return FALSE
  endfor
  return TRUE
end

```

Figure 3. *compatible* checks if two two states are approximately equivalent

The check for $\mathbf{E}(s_i) \sim \mathbf{E}(s_j)$ is performed by the function *differentExpMean*, described in Figure 4, which uses statistical hypothesis testing. The function actually checks if the means $1/\mathbf{E}(s_i)$ and $1/\mathbf{E}(s_j)$ of two exponential distributions are different. Given two exponential distributions with means θ_1 and θ_2 we want to check if $\theta_1 = \theta_2$ against

the fact fact that $\theta_1 \neq \theta_2$. This is equivalent to checking $\frac{\theta_1}{\theta_2} = 1$ against the fact that $\frac{\theta_1}{\theta_2} \neq 1$. In statistical terms we call $\frac{\theta_1}{\theta_2} = 1$ as the null hypothesis (denoted by H_0) and $\frac{\theta_1}{\theta_2} \neq 1$ as the alternate hypothesis (denoted by H_a). To test the hypothesis H_0 against H_a we draw n_1 samples, say x_1, x_2, \dots, x_{n_1} , from the exponential distribution with mean θ_1 and n_2 samples, say y_1, y_2, \dots, y_{n_2} , from the exponential distribution with mean θ_2 . We estimate θ_1 and θ_2 by $\hat{\theta}_1 = \frac{\sum_{i=1}^{n_1} x_i}{n_1}$ and $\hat{\theta}_2 = \frac{\sum_{i=1}^{n_2} y_i}{n_2}$ respectively. Then we use the ratio $\frac{\hat{\theta}_1}{\hat{\theta}_2}$ to check H_0 against H_a as follows:

We can say that x_1, x_2, \dots, x_{n_1} are random samples from the random variables X_1, X_2, \dots, X_{n_1} where each X_i has an exponential distribution with mean θ_1 . Similarly, y_1, y_2, \dots, y_{n_2} are random samples from the random variables Y_1, Y_2, \dots, Y_{n_2} where each Y_i has an exponential distribution with mean θ_2 . Then it can be shown by methods of moment generating function that the random variables $\frac{2\sum X_i}{\theta_1}$ and $\frac{2\sum Y_i}{\theta_2}$ have $\chi^2(2n_1)$ and $\chi^2(2n_2)$ distributions respectively. This implies that the ratio $\frac{(2\sum X_i)/(2n_1\theta_1)}{(2\sum Y_i)/(2n_2\theta_2)}$ or $\frac{\sum X_i/n_1}{\theta_1} / \frac{\sum Y_i/n_2}{\theta_2}$ has F distribution with $(2n_1, 2n_2)$ degrees of freedom. Assuming that H_0 holds $\frac{\sum X_i/n_1}{\sum Y_i/n_2}$ has $F(2n_1, 2n_2)$ distribution. Let us introduce the random variables Θ_1 and Θ_2 where $\Theta_1 = \frac{\sum X_i}{n_1}$ and $\Theta_2 = \frac{\sum Y_i}{n_2}$. Our experimental value of $\frac{\hat{\theta}_1}{\hat{\theta}_2}$ gives a random sample from the random variable $\frac{\Theta_1}{\Theta_2}$. Let the random variable $Z = \frac{\Theta_1}{\Theta_2} \theta_2$. Then Z has F distribution with $(2n_1, 2n_2)$ degrees of freedom. Given $\theta_1 = \theta_2$, from Chebyshev's inequality, we get

$$\text{Prob} \left[\left| \frac{\Theta_1}{\Theta_2} - \mu \right| \geq \frac{\sigma}{\sqrt{\alpha}} \right] = \text{Prob} \left[|Z - \mu| \geq \frac{\sigma}{\sqrt{\alpha}} \right] \leq \alpha$$

where $\mu = \frac{n_2}{n_2-1}$ is the mean of $F(2n_1, 2n_2)$ and $\sigma = \sqrt{\frac{n_2^2(n_1+n_2-1)}{n_1(n_2-1)^2(n_2-2)}}$ its standard deviation. Thus, taking $\check{r} = \mu - \frac{\sigma}{\sqrt{\alpha}}$ and $\hat{r} = \mu + \frac{\sigma}{\sqrt{\alpha}}$, we get

$$\text{Prob} \left[\check{r} \leq \frac{\Theta_1}{\Theta_2} \leq \hat{r} \right] > 1 - \alpha \quad (1)$$

If $\frac{\hat{\theta}_1}{\hat{\theta}_2} > 1$ then we calculate the probability of our observation given $\theta_1 = \theta_2$, called the *p-value*, as

$$\text{p-value} = \text{Prob} \left[\frac{\Theta_1}{\Theta_2} > \frac{\hat{\theta}_1}{\hat{\theta}_2} \right] = \text{Prob} \left[Z > \frac{\hat{\theta}_1}{\hat{\theta}_2} \right]$$

Similarly, if $\frac{\hat{\theta}_1}{\hat{\theta}_2} < 1$, the *p-value* is given by

$$\text{p-value} = \text{Prob} \left[\frac{\Theta_1}{\Theta_2} < \frac{\hat{\theta}_1}{\hat{\theta}_2} \right] = \text{Prob} \left[Z < \frac{\hat{\theta}_1}{\hat{\theta}_2} \right]$$

If the calculated *p-value* in both cases together is less than α we say we have enough evidence to reject the null hypoth-

esis $\theta_1 = \theta_2$. This is equivalent to say that we reject H_0 if $\frac{\hat{\theta}_1}{\hat{\theta}_2} \notin [\check{r}, \hat{r}]$.

```

algorithm differentExpMeans
Input:  $\hat{\theta}_1, n_1, \hat{\theta}_2, n_2, \alpha$ 
Output: boolean
begin
  if  $n_1 = 0$  or  $n_2 = 0$  then
    return FALSE
  return  $\frac{\hat{\theta}_1}{\hat{\theta}_2} \notin [\check{r}, \hat{r}]$ 
end

```

Figure 4. *differentExpMeans* checks if two estimated exponential means are different; the parameter α is used in calculating \check{r} and \hat{r}

The check for $\mathbf{P}(s_i, a) \sim \mathbf{P}(s_j, a)$ is performed by the function *differentBerMeans* (see Figure 5) using Hoeffding bounds similar to that in [6]. The method checks if the means p_1 and p_2 of two Bernoulli distributions are statistically different or not. If f_1 tries are 1 out of n_1 tries from a Bernoulli distribution with mean p_1 and f_2 tries are 1 out of n_2 tries from a Bernoulli distribution with mean p_2 , then we say that p_1 and p_2 are statistically same if

$$\left| \frac{f_1}{n_1} - \frac{f_2}{n_2} \right| < \sqrt{\frac{1}{2} \log \frac{2}{\alpha}} \left(\frac{1}{\sqrt{n_1}} + \frac{1}{\sqrt{n_2}} \right)$$

Note that it is possible to use other tests, such as multinomial test [16], to compare two means of Bernoulli distributions.

```

algorithm differentBerMeans
Input:  $\hat{p}_1, n_1, \hat{p}_2, n_2, \alpha$ 
Output: boolean
begin
  if  $n_1 = 0$  or  $n_2 = 0$  then
    return FALSE
  return  $|\hat{p}_1 - \hat{p}_2| > \sqrt{\frac{1}{2} \log \frac{2}{\alpha}} \left( \frac{1}{\sqrt{n_1}} + \frac{1}{\sqrt{n_2}} \right)$ 
end

```

Figure 5. *differentBerMeans* checks if two estimated Bernoulli means are different

3.4. Complexity

The worst case running complexity of the algorithm is cubic in the sum of the length of all samples. However, in our experiments we found that the running time grows almost linearly with the sum of length of sample lengths. The parameter α influences the size of the sample needed for

converging on the right model. The exact dependence of sample size on α is an open problem that needs investigation.

4. Learning in the Limit

In order to prove correctness of our algorithm, we need to show that the $CTMC_L$ that the learning algorithm produces is eventually equivalent to the model that was used to generate the samples. Our proof proceeds in two steps. First we show that the learning algorithm will eventually be presented what is usually called a *structurally complete* sample. A structurally complete sample I^+ is a multi-set of traces such that the traces visit every (reachable) state and every transition. More formally, for every state s of the target $CTMC_L$, there is a trace $\tau \in I^+$ such that s is one of the states visited when trace τ was produced, and for every (reachable) transition (s, a) there is a trace $\tau \in I^+$ such that (s, a) is traversed by τ . Observe that if $I^+ \subseteq I'^+$ and I^+ is structurally complete then I'^+ is also structurally complete. The second step of the proof involves showing that if we keep adding samples to a structurally complete set, then we will eventually learn the right $CTMC_L$. These two steps together show that our algorithm will learn the target $CTMC_L$ in the limit [8].

The first thing to observe that for any $CTMC_L \mathcal{M}$ there is a finite structurally complete sample set. Let Γ be a structurally complete sample set and let $\mathcal{F} = (\mathcal{M}, p)$ be a $CTMC_L^f$ (with any stopping probability p). Now observe that for any $\tau \in \Gamma$, $p = Prob_{\mathcal{F}}(\tau, s_0)$ is finite and non-zero. Thus, the probability that τ is not among the first k samples generated by \mathcal{F} is $(1-p)^k$, and this tends to 0 as k increases. Hence, every string in Γ is eventually generated, and so the sample given to the learning algorithm is eventually structurally complete.

The main challenge in the proof of correctness is to show that once we have a structurally complete sample, we will eventually learn the right $CTMC_L$. In what follows, we simply assume that whenever we refer to a sample I^+ , we mean that I^+ is structurally complete. Observe that for a (structurally complete) sample I^+ , the right $CTMC_L$ is one that results from merging equivalent states of $PCTMC(I^+)$. However, since we can only check compatibility (and not exact equivalence) the only errors the algorithm makes can result when we check the compatibility of two states. There are two types of errors in this context.

1. Type I error : *compatibility* returns false when two states are actually equivalent, and
2. Type II error : *compatibility* returns true when two states are not equivalent

Our goal is to reduce these two errors as much as possible. We show that as $s = |I^+|$ goes to infinity, the global contribution of these two errors tend to zero. Observe that, if t is

the number of states in $PCTMC(I^+)$, then t cannot grow as fast as s does. If m be the number of states in the target $CTMC_L^f$, then the number of merges performed by the algorithm before giving the correct $CTMC_L^f$ is $t - m$. Further recall that the p -value of the tests performed by the functions *differentExpMeans* and *differentBerMeans* is at most α . Hence, global Type I error, e_α is bounded by $\alpha(|\Sigma| + 1)t$. This error can be made negligible and independent of the size of the $PCTMC(I^+)$ by taking $\alpha = kt^{-1}$ for some very small constant k .

Thus, by making α small we can ensure that the learning algorithm always merges equivalent states. Then the errors of the learning algorithm can be confined to those resulting from merging inequivalent states. In the absence of Type I errors, the learning algorithm always outputs a $CTMC_L^f$, whose states form a partition of the target $CTMC_L^f$. Thus an upper bound on Type II errors is given by the probability that an error occurs when comparing two states of the target $CTMC_L^f$. Taking β to be the probability of merging two non-equivalent states, we get the Type II error $e_\beta \leq \frac{1}{2}\beta m(m-1)(|\Sigma| + 1)$. Thus if we show that β tends to 0 as the sample size grows, then we know that the algorithm will eventually not make any errors.

Observe that the probability of merging a pair of non-equivalent states is bounded by the probability of either *differentExpMeans* or *differentBerMeans* returning TRUE when the actual means are different. Hence, in order to show that the learning algorithm eventually gives the right answer, we need to show that the probability that *differentExpMeans* and *differentBerMeans* make an error when the means are different tends to 0. We will consider each of the procedures separately and bound their error.

Case *differentExpMeans*: Assume that $r = \frac{\theta_2}{\theta_1} \neq 1$, i.e., $\mathbf{E}(s_i) \neq \mathbf{E}(s_j)$. Recall that for a given α , taking $\check{r} = \mu - \frac{\sigma}{\sqrt{\alpha}}$ and $\hat{r} = \mu + \frac{\sigma}{\sqrt{\alpha}}$, where μ is the mean and σ the standard deviation of $F(2n_1, 2n_2)$, we say that an observation $\frac{\theta_1}{\theta_2}$ provides evidence for $\mathbf{E}(s_i) \sim \mathbf{E}(s_j)$ when $\frac{\theta_1}{\theta_2} \in [\check{r}, \hat{r}]$. Now, we have

$$\begin{aligned} \text{Prob} \left[\check{r} \leq \frac{\theta_1}{\theta_2} \leq \hat{r} \right] &= \text{Prob} \left[\check{r}r \leq \frac{\theta_1}{\theta_2} \frac{\theta_2}{\theta_1} \leq \hat{r}r \right] \\ &= \text{Prob} [\check{r}r \leq Z \leq \hat{r}r] \end{aligned}$$

where, Z has distribution $F(2n_1, 2n_2)$. Once again by Chebyshev's inequality, we know that

$$\text{Prob} \left[|Z - \mu| \geq \frac{\sigma}{\sqrt{\beta}} \right] \leq \beta$$

Thus, if $r\check{r} \geq \mu + \frac{\sigma}{\sqrt{\beta}}$ or $r\hat{r} \leq \mu - \frac{\sigma}{\sqrt{\beta}}$ then

$$\text{Prob} \left[\check{r} \leq \frac{\theta_1}{\theta_2} \leq \hat{r} \right] = \text{Prob} [\check{r}r \leq Z \leq \hat{r}r] \leq \beta$$

Taking

$$\beta = \left(\frac{\sigma\sqrt{\alpha}}{|r - 1|\mu\sqrt{\alpha} + r\sigma} \right)^2$$

we observe that if $r < 1$ then $r\check{r} \geq \mu + \frac{\sigma}{\sqrt{\beta}}$ and if $r \geq 1$ then $r\hat{r} \leq \mu - \frac{\sigma}{\sqrt{\beta}}$. Finally, plugging in the values of μ , σ and α , we observe that β tends to 0.

Case *differentBerMeans*: Let $\mathbf{P}(s_i, a) = p_1 \neq p_2 = \mathbf{P}(s_j, a)$. Let F_1 be a random variable that is the mean of n_1 Bernoulli trials with mean p_1 and F_2 the mean of n_2 Bernoulli trials with mean p_2 . Recall that we say $\mathbf{P}(s_i, a) \sim \mathbf{P}(s_j, a)$ if some observation \hat{f}_1 of variable F_1 and observation \hat{f}_2 of F_2 are such that $|\hat{f}_1 - \hat{f}_2| < \epsilon$ where

$$\epsilon = \sqrt{\frac{1}{2} \log \frac{2}{\alpha}} \left(\frac{1}{\sqrt{n_1}} + \frac{1}{\sqrt{n_2}} \right)$$

We will try to bound the probability that this happens. Observe that $E(F_1 - F_2) = p_1 - p_2$ and

$$\begin{aligned} \text{Var}(F_1 - F_2) &= \text{Var}(F_1) + \text{Var}(F_2) \\ &= \frac{p_1(1-p_1)}{n_1} + \frac{p_2(1-p_2)}{n_2} \geq \frac{1}{4n_1} + \frac{1}{4n_2} \end{aligned}$$

Now $\beta = \text{Prob}(|F_1 - F_2| < \epsilon) < \text{Prob}(|(F_1 - F_2) - (p_1 - p_2)| > |p_1 - p_2| - \epsilon)$. By Chebyshev's inequality, $\beta \leq B$ where

$$B = \begin{cases} \frac{\text{Var}(F_1 - F_2)}{(|p_1 - p_2| - \epsilon)^2} & \text{if } \epsilon < |p_1 - p_2| \text{ and} \\ & \text{Var}(F_1 - F_2) < ((p_1 - p_2) - \epsilon)^2 \\ 1 & \text{otherwise} \end{cases}$$

First observe that n_1 and n_2 grow linearly as $s = |I^+|$ increases, and so even if α is kt^{-1} as needed in order to eliminate Type I errors, since t is bounded by s , ϵ tends to 0 as s grows. Next $\text{Var}(F_1 - F_2)$ also tends to 0 as s grows. Hence B vanishes with s , proving that in the limit Type II errors are eliminated.

5. Tool and Experiments

We have implemented the learning algorithm in Java as a sub-component of the tool VESTA (Verification based on Statistical Analysis).² The tool takes a multi-set of examples generated from the simulation of a system having an unknown $CTMC_L$ model. Based on these examples the tool learns the underlying $CTMC_L$ for a given value of α . The learned model can be used for verification of CSL formulas either using the statistical model-checker of VESTA [21] or other model-checkers such as PRISM [17], ETMCC [12], Ymer [24] etc. We tested the performance of our tool on several $CTMC_L$ models. For each $CTMC_L$ model we performed discrete-event simulation to get a large number of examples and then learned a $CTMC_L$ based on these examples. Finally we checked if the learned $CTMC_L$ is equivalent as the original $CTMC_L$, that generated the examples. We found that the learned $CTMC_L$ is equivalent to the original $CTMC_L$ in all our experiments provided that the number of samples is large enough. In all our experiments we assumed that the set of atomic propositions at any state

² Available from <http://osl.cs.uiuc.edu/~ksen/vesta/>

are same. This is assumed to show the working of the statistical tests. If we take atomic propositions into consideration then the learning becomes faster because atomic propositions will be sufficient in distinguishing certain states. We next report the results of our experiments performed on a Pentium III 1.2GHz laptop with 620 MB SDRAM.

Symmetric CTMC The $CTMC_L$ in Figure 6, which is carefully selected, contains four states. The probability of taking the edges labeled a and b from the states 0 and 2 are same; however, the total rates at which transitions take place from the two states are different. Therefore, to distinguish these two states comparison of the total rates is required. Similar is the case for states 1 and 3. On the other hand, the total rates for the states 0 and 1 are same; however, the probability of taking the edges a and b are different which is used to distinguish the two states. The same is true for the states 2 and 3. Thus this example shows the effectiveness of both the functions *differentBerMeans* and *differentExpMeans* during learning. We use the $CTMC_L$ to generate samples which are then used to learn a $CTMC_L$. We found that for more than 600 samples and $\alpha = 0.0001$ the $CTMC_L$ learned is same as the original $CTMC_L$.

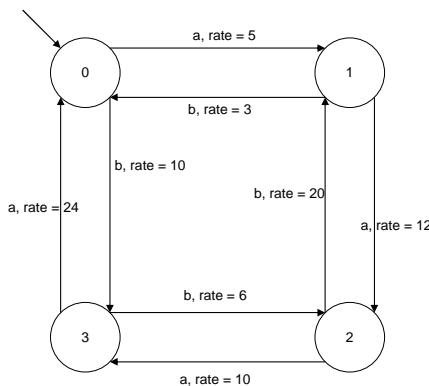


Figure 6. Symmetric CTMC

Triple Modular Redundant System Our next example is the $CTMC_L$ in Figure 7 representing the model of a *Triple Modular Redundant System* (TMR). The example is taken from [10, 2]. We ignore the atomic propositions that are true at each state to show the effectiveness of our statistical test. Although we generated the samples through the discrete event simulation of the $CTMC_L$ in Figure 7, we can as well assume that the samples are coming from the actual running system. In figure 7 we plot the average number of states in the learned $CTMC_L$ and time taken in learning against the number of samples used in learning. The number of states converges to five when the sample is large enough. The time taken for learning grows almost linearly with the number of samples. With $\alpha = 0.0001$ and 1000 samples the algorithm learns the same $CTMC_L$ in less than 2 sec-

onds. For small number of samples, due to lack of sufficient information, the algorithm tends to generalize more resulting in less number of states in the learned $CTMC_L$.

Tandem Queuing Network In this more practical example we considered a M/Cox₂/1-queue sequentially composed with a M/M/1-queue. This example is taken from [13]. For $N = 3$, where N denotes the capacity of the queues, the algorithm learned a $CTMC_L$ model with 15 states. However, the number of samples required in this case to learn the underlying $CTMC_L$ is quite large (around 20,000). This particular experiment suggests that learning underlying $CTMC_L$ for large systems may require a large number of samples. Therefore, a more effective technique would be to verify the approximate model learnt from small number of samples. However, because the model learnt is approximate, the result of verification would also be approximate. This suggests that the confidence in verification should be quantified reasonably. How to do such quantification remains an open problem.

6. Conclusion and Future Work

We have presented a novel machine-learning algorithm to learn the underlying edge-labeled continuous-time Markov chains of deployed stochastic systems for which we do not know the model before-hand. An important aspect of the learning algorithm is that it can learn a formal stochastic model from the traces generated during testing or executing the deployed system. The learnt $CTMC_L$ can be used to verify the deployed systems using existing probabilistic model-checking tools. Moreover, one can also check if the learnt model is F -bisimilar [2, 3] to the model given in a specification. This allows us to check if the deployed system correctly implements a specification with respect to a set of formulas F . Finally, we provide an implementation of the algorithm which can be used with various other tools.

One of the limitations of our work is that it may not scale up for systems having large underlying CTMC model. Therefore, one needs to develop techniques that can perform approximate verification as the model is learnt. The accuracy of such verification technique should increase with the increase in the number of samples. The difficult part in developing such an approach is to correctly quantify the confidence (or accuracy) in verification. Such a technique will make verification of “black-box” systems a very practical approach that can co-exist with testing based on discrete event simulation.

Acknowledgement

The work is supported in part by the DARPA IPTO TASK Award F30602-00-2-0586, the DARPA/AFOSR MURI Award F49620-02-1-0325, the ONR Grant N00014-02-1-0715, and the Motorola Grant MOTOROLA RPS #23 ANT. Our work has benefitted considerably from our collaboration with Abhay Vardhan

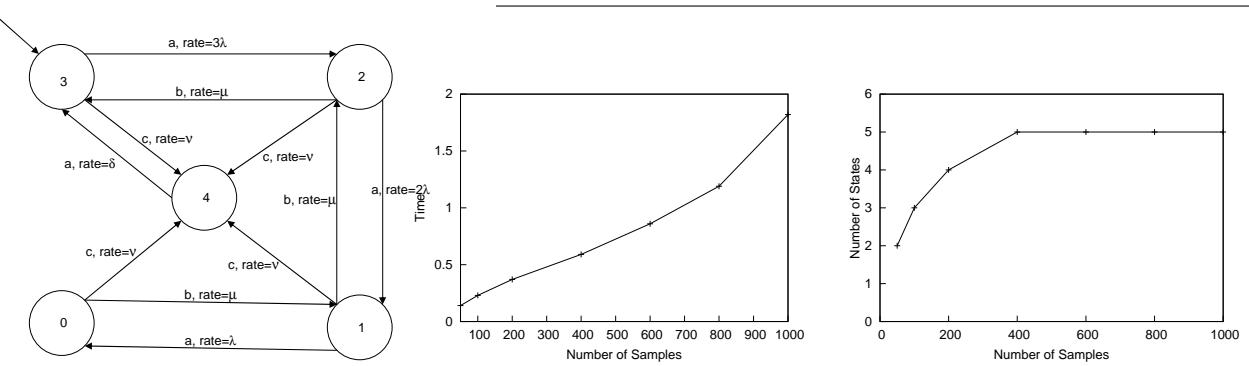


Figure 7. Learning TMR

on “learning to verify” framework for verifying infinite state systems. We would like to thank Tom Brown for giving us feedback on a previous version of this paper.

References

- [1] D. Angluin. Learning regular sets from queries and counterexamples. *Infor. and Comp.*, 75(2):87–106, 1987.
- [2] C. Baier, B. Haverkort, H. Hermanns, and J. Katoen. Model checking continuous-time Markov chains by transient analysis. In *Computer Aided Verification (CAV’00)*, volume 1855 of *LNCS*, pages 358–372. Springer, 2000.
- [3] C. Baier, J. Katoen, H. Hermanns, and B. Haverkort. Simulation for continuous-time Markov chains. In *13th International Conference on Concurrency Theory (CONCUR’02)*, volume 2421 of *LNCS*, pages 338–354. Springer, 2002.
- [4] G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, and M. A. Marsan. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.
- [5] L. Baum, T. Petrie, G. Soules, and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *Annals of Mathematical Statistics*, 41(1):164–171, 1970.
- [6] R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *International Colloquium Grammatical Inference and Applications (ICGI’94)*, volume 862 of *LNCS*. Springer, 1994.
- [7] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS’03)*, volume 2619 of *LNCS*, pages 331–346.
- [8] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.
- [9] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS’02)*, volume 2280 of *LNCS*, pages 357–371, 2002.
- [10] B. Haverkort. *Performance of Computer Communication Systems: A Model-Based Approach*. Wiley, 1998.
- [11] H. Hermanns, U. Herzog, and J. Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 274(1–2):43–87, 2002.
- [12] H. Hermanns, J. P. Katoen, J. Meyer-Kayser, and M. Siegle. A tool for model-checking Markov chains. *Software Tools for Technology Transfer*, 4(2):153–172, 2003.
- [13] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi-terminal binary decision diagrams to represent and analyse continuous-time Markov chains. In *Workshop on the Numerical Solution of Markov Chains (NSMC’99)*, 1999.
- [14] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
- [15] R. V. Hogg and A. T. Craig. *Introduction to Mathematical Statistics*. Macmillan, New York, NY, USA, 1978.
- [16] C. Kermorvant and P. Dupont. Stochastic grammatical inference with multinomial tests. In *Grammatical Inference: Algorithms and Applications*, volume 2484 of *Lecture Notes in Artificial Intelligence*. Springer, 2002.
- [17] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS’02)*, volume 2324 of *LNCS*, pages 200–204.
- [18] W. Nelson. *Applied Life Data Analysis*. Wiley, 1982.
- [19] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pages 49–61. 1992.
- [20] D. Ron, Y. Singer, and N. Tishby. On the learnability and usage of acyclic probabilistic finite automata. *Journal of Computer and System Sciences*, 56(2):133–152, 1998.
- [21] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *16th conference on Computer Aided Verification (CAV’04)*, LNCS (To Appear). Springer, July 2004.
- [22] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton, 1994.
- [23] W. Wei, B. Wang, and D. Towsley. Continuous-time hidden Markov models for network performance evaluation. *Performance Evaluation*, 49(1–4):129–146, September 2002.
- [24] H. L. S. Younes, M. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking: An empirical study. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS’04)*, volume 2988 of *LNCS*. Springer, 2004.

Towards a Hierarchical Taxonomy of Autonomous Agents *

Predrag T. Totic and Gul A. Agha
Open Systems Laboratory, Department of Computer Science
University of Illinois at Urbana-Champaign, USA
E-mail: {p-totic, agha}@cs.uiuc.edu

Abstract – *Autonomous agents have become an influential and powerful paradigm in a great variety of disciplines, from sociology and economics to distributed artificial intelligence and software engineering to philosophy. Given that the paradigm has been around for awhile, one would expect a broadly agreed-upon, solid understanding of what autonomous agents are and what they are not. This, however, is not the case. We therefore join the ongoing debate on what are the appropriate notions of autonomous agency. We approach agents and agent ontology from a cybernetics and general systems perspective, in contrast to the much more common in the agent literature sociology, anthropology and/or cognitive psychology based approaches. We attempt to identify the most fundamental attributes of autonomous agents, and propose a tentative hierarchy of autonomous agents based on those attributes.*

Keywords: Smart Agents, Intelligent Systems, Intelligent and Soft Computing - Systems and Applications

1 Introduction and Motivation

Autonomous agents have become a powerful paradigm in a great variety of disciplines, from sociology and economics to distributed artificial intelligence and software engineering to cognitive sciences to philosophy. While different disciplines have different needs and may have different notions of *agents*, the agents in economics and those in distributed artificial intelligence (DAI), for example, nonetheless tend to share most of the fundamental properties. Given that the paradigm has been around for awhile, one would expect a relatively solid understanding of what autonomous agents are, and what they are not. This, however, is not at all the case.

Is there, then, at least a reasonably unified and broadly agreed upon notion of *autonomous agency* among the computer scientists? The answer is still ‘No.’ In particular, the notion of autonomous agency in open distributed computing environments (e.g., [2, 6]), while certainly sharing some of the properties, does not coincide with the corresponding standard notion of agency in artificial intelligence (e.g., [18]). One would hope that understanding this gap

can assist in bridging it, thereby enhancing the ability of the software system designers to meet the requirements of various AI and other applications more effectively and easily by readily identifying and efficiently building the required additional functionality (“agent capabilities”) on the top of the existing open distributed agent-based (or even merely object-based) software infrastructures.

Herewith, we attempt to contribute to the ongoing debate on what are the appropriate notions of *autonomous agency*. Instead of proposing a single such prescriptive (and therefore necessarily also restrictive), “one size fits all” definition of autonomous agents, we propose an entire hierarchy of agents, from simpler (reactive situated agents) towards quite complicated and capable of human-like complex cognitive tasks (deliberative, intelligent agents). The proposed hierarchy, rather than being based on any particular school of thought in artificial intelligence and cognitive sciences, is chiefly based on ideas and paradigms from other scientific disciplines - mainly *cybernetics* [25] and *systems science* [14, 15]. We argue that learning from other, non-AI and non-cognitive disciplines such as cybernetics or biology can provide some critical, yet thus far for the most part missing, ingredients in building successful and complete theories of artificial autonomous agents and multi-agent systems. This work is intended to be a modest step in that direction.

2 What Are Autonomous Agents?

It has become common to define an appropriate notion of agency by specifying the *necessary attributes* that all agents of the particular kind one has in mind are required to share (e.g., [10, 16, 18]). There has been much of debate, however, what set of properties exactly qualifies an entity, such as a single human decision maker, a firm in the market, a computer program, a robot or an unmanned autonomous vehicle, for an *autonomous* or an *intelligent* agent. Influential position papers, such as [26] for intelligent agents or [10] for autonomous agents, while trying to clarify and unify the terminology, and propose agent taxonomies, also illustrate the heterogeneity and lack of agreement on the definition and the required (as opposed to optional) properties even in the case of autonomous agents that are restricted to computer

*0-7803-8566-7/04/\$20.00 © 2004 IEEE.

programs alone (which disallows, say, humans or social insects).

It has been observed that the main division line is the one that separates the (purely) *reactive agents* [17, 18] from the more complex, capable of cognitive-like behaviors *deliberative agents* [16, 18, 26]. A reactive agent is one that is coupled to the environment and is capable of being affected by, and perhaps in turn also affecting, the environment. It need not be capable of cognitive tasks such as learning, planning or reasoning. It need not have any complicated internal structure, or any capability of complex correlations between its internal states and the states of the outer world (“symbolic representations”); it uses a little or no memory, etc.

In contrast, a deliberative agent is much more complex in terms of its internal structure, is typically capable of creating and working with abstract representations of a complex outer world (e.g., by performing planning, reasoning and/or learning tasks), has some sense of its purpose (tasks, goals, utilities), usually is pro-active and adaptable, etc. Much of research in the main-stream artificial intelligence (AI) over the past twenty or more years has been focused on the design problem of such artificial deliberative agents, capable of acting in complex environments and autonomously pursuing their complex goals or tasks in such environments (see, e.g., [6, 16, 18, 24, 26] and references therein).

Herein, we attempt to hierarchically classify agents based on their complexity in terms of their capabilities and functionalities, not on (models of) agents’ internal structure. An agent is more sophisticated than another, if it is capable of more complex behaviors observable by an outside observer. This natural functionalist, behaviorist and systems theory oriented approach, however, does not seem very common in the mainstream agent literature.

Some of the most frequently encountered general properties of agents found in the literature include reactivity, pro-activeness, ability to execute autonomously, goal-orientedness or goal-drivenness, a capability of sensing the environment and being affected by the environment, a capability of affecting the environment, sociability, ability to communicate, persistence, purposefulness, and ability to learn and/or reason about the world.

Not all the agents have to possess all of the above mentioned properties, of course. We shall make an attempt, however, to identify those properties that are *necessary* for autonomous agents of a desired level of complexity.

In case of the computer programs, being capable of autonomous execution, that is, an execution that is not (entirely) controlled from the outside, seems to be the most natural requirement for any notion of autonomous agency. However, a question then arises, is this enough? For instance, a *finite state machine (FSM)* executes autonomously (and reactively, inasmuch as the ability of an agent to be affected by the environment suffices for reactivity), but we find it hard to consider *individual FSMS* an appropriate abstraction of autonomous agents. On the other hand, a *coupled* finite automata model has been proposed as an abstraction of reac-

tive situated agents (e.g., [17]). We shall discuss in some detail what we consider to be the necessary attributes of autonomous agency, as well as propose a hierarchy of agents in terms of the attributes they possess, in *Section 4*.

3 A Systems Approach To Agents

Most approaches to classifying various types of (natural as well as artificial) agents are based on specifying the necessary *attributes* of a particular kind of agents, as in, e.g., [10]. We adopt this general approach, as well. However, we also try to be more specific as to *what kinds of attributes* we allow. Tools from other cognitive disciplines, such as psychology, anthropology and sociology, have been liberally applied to characterize the fundamental properties, and therefore the very nature, of various artificial agent systems. In particular, software and robotic agents have been generously ascribed properties that characterize anthropomorphic cognition, such as beliefs, desires, intentions, emotions, etc. One of the most successful examples of such approach are the BDI agent paradigm and architectures [16].

However, we see some potential conceptual and practical problems with assigning too liberally human (cognitive or other) attributes to a piece of software or a robot. In scientific and engineering modeling, the very purpose of a *model* is to be intrinsically simpler, and therefore more amenable to analysis, than the phenomenon being modeled. But when the attributes of beliefs, intentions, emotions, and the like are ascribed to, for instance, a software agent system with individual agents of a fairly modest complexity, it seems that exactly the opposite is the case. While there is some justification in correlating, for instance, how artificial agents represent and interact with complex, partially observable environments and tasks to how humans act (reason, learn, represent knowledge, etc.) with respect to their tasks and environments, there are also certain dangers in this approach. For, after all, robots and software agents are not human, and (unless one believes in the Strong AI hypothesis [19]) perhaps cannot ever be made very human-like in terms of their cognitive capabilities. Furthermore, representing and reasoning about relatively simple software agents encountered in many software engineering applications in terms of highly complex capabilities of human-like cognition seems to be an “overkill”, in that the complexity of the model may end up considerably exceeding the sophistication of the modeled.

Another problem with attributing various anthropomorphic features to artificial agents emerges once different types of such agents are compared and contrasted with one another. Software agents, robots and other types of artificial agents are man-designed engineering systems. They should be characterized, studied, compared and contrasted to one another in terms of how they as systems *behave*, not what “mental states” or “beliefs” or “desires” or “emotions” their designer attributes to them. Whether an agent is reactive or adaptable can be, in general, verified by an outside observer that is independent of the agent. What are the belief or desire or emotional states of an agent, on the other hand, cannot.

We shall propose in the sequel a less cognition-oriented, and less anthropomorphic, approach to modeling, classifying and understanding various types of (artificial) autonomous agents and multi-agent systems (MAS). In particular, our approach, instead of cognitive psychology, draws more analogies and paradigms from cybernetics [25] and systems science [14, 15] on one, and biology and natural evolution [9], on the other hand. We argue that this approach fairly naturally leads to various possible hierarchical classifications of autonomous agents, and we propose one such general and broad agent hierarchy.

In particular, instead of comparing various agents in terms of their sophistication by chiefly comparing the complexities of agents' internal representations or "logics", we adopt a cybernetics-inspired approach based on the "black box" abstraction, and consider what kind of properties an agent needs in order to be able to do certain things, or function a certain way. We view an agent system "not a thing, but a list of variables" [5] and relations among those variables. Moreover, to understand an autonomous agent, one has to also understand this agent's environment, as well as various loops (e.g., feed-forward or feedback) that determine how this agent interacts with its environments. Thus our emphasis is on a functionalist, behavioral aspects of agency, and an agent is viewed as a black box whose inner structure (such as beliefs, desires, emotions, etc.) may or may not be accessible or understood, but it is *the interaction of this black box system with the outside world*, mechanisms for that interaction, and observable behavioral consequences of that interaction that are given the "first class" status (see, e.g., [4, 5]).

4 An Agent Hierarchy: From Reactive Towards Deliberative

We now discuss in some detail what are the critical, *necessary* (as opposed to optional) attributes that characterize most known autonomous agents, biological and computational alike. The most elementary attributes of such agents can be expected to be those properties that characterize any *autonomous system* in general. Once a couple of additional attributes that characterize virtually all agents are added, we arrive at a *weak* notion of autonomous agency. Subsequently, some additional properties will be identified that, we argue, characterize nearly all autonomous agents found in AI and DAI. An agent that possesses each of these attributes, as well as those of weakly autonomous agents, we shall call *strongly autonomous*. Finally, one more property will be identified that is absolutely necessary for any (however weak) notion of intelligence. Thus this list of system properties, each to at least some degree observable or testable by an observer external to the system, will implicitly define a tentative *natural hierarchy* of autonomous agents. In addition to similar attempts at classifying various types of agents (e.g., [10, 26]), our approach is also motivated by the general systems theory, and, in particular, by epistemological hierarchies of (*general*) systems, as in, e.g., [15].

The minimal notion of autonomy is the requirement that an entity (at least partially) controls its own internal state. Some degree of control¹ of one's internal state indeed appears necessary for autonomous agency, as well - but it is by no means sufficient. In addition to control over its *internal state*, an autonomous system ought to have at least some degree of control over its *behavior*. In case of a computer program (that is, a software agent), this means autonomous execution. If some autonomous control of a software system's state and execution were all it takes for such a system to be an autonomous agent, then the distinction between software agents and arbitrary computer programs would be rather blurred, and (*almost*) all programs would "qualify" for autonomous agents (see, e.g., discussions in [10, 17]). This is clearly undesirable. The question arises, what is missing - what additional requirements need to be imposed on an arbitrary computer program so that such a program can be considered a legitimate software agent?

Agents cannot be understood in isolation from the environment in which they are embedded [10]. This implies that, in order to develop a meaningful model of an agent, we need (a) an appropriate model of the environment, and (b) a model of the agent's *interaction* with the environment.

Regardless of the nature and mechanisms of this interaction between an agent and its environment (where the environment may also include other agents), there would be no point to any such interaction if it were not able to *affect* either the agent, or the environment outside of the agent, or, most often in practice, *both*.

Consequently, we consider *reactivity* (or what is called "*responsiveness*" in [26]) to be another necessary attribute of any notion of autonomous agency, as the agent has to be able to (1) notice changes in the environment, (2) appropriately respond to those changes, and (3) affect what input or stimuli it will receive from the environment in the future. Hence, the necessary attributes for any reasonable notion of autonomous agency identified thus far are (i) some control of one's internal state and execution, and (ii) reactivity as a prerequisite for the agent-environment interactions that, in general, may affect both the agent and the environment.

Any "proper" computational or biological autonomous agent can also be expected to be at least somewhat *persistent*, that is, to "live on" beyond completing a single task on a single occasion. In case of software agents, persistence makes an agent different from say a *subroutine* of a computer program whose "turning on and off" is controlled from outside of that subroutine (see, e.g., [10]). This necessity of some form of persistence is evidently strongly related to the most basic requirement of (*weakly*) *autonomous agency*, namely, that an agent ought to have some degree of control of its internal state and behavior.

¹Full and exclusive control of one's internal state, if understood in the sense of that "nothing from the outside" can affect the entity's state, is clearly not desirable in case of agents, as one would like the agent to be able to be effected by its environment.

We summarize below our notion of *weakly autonomous agency* (*WAA*)² in terms of the necessary agent attributes:

$$\begin{aligned} \text{weak autonomous agency} &\approx \text{control of own state} \\ &+ \text{reactivity} + \text{persistence} \end{aligned}$$

Hence, at the bottom level of the emerging hierarchy of autonomous agents, we find purely reactive embedded (or situated) agents [17]. Such agents can be appropriately abstracted via finite state machines (deterministic case) or discrete Markov chains (probabilistic case). A combination of reactivity and persistence characterizes many of both the simplest life forms and simple artificial agents. When some degree of control of the agent's internal state and behavior is also present, one arrives at what we shall call herewith *weakly autonomous agency* (*WAA*). We suggest the actor model of distributed computing [1, 2] to be a canonical example of *WAA* among the software agents.

One common feature found in all or nearly all interesting autonomous agents, biological and computational alike, is some form of *goal-orientedness* or *goal-drivenness*. In case of the living organisms, the highest level driving mechanisms are the instincts of *survival* and *reproduction*. The single most fundamental instinct in all of known life forms (to which an appropriate notion of an instinct can be ascribed at all) is that of survival. Indeed, the instinct of reproduction is related to the survival of the species or, perhaps, of the particular genes and gene patterns, as opposed to the "mere" survival of the individual organisms [9]. At lower levels, the driving mechanisms - finding food or a sexual partner - are those that are expected to provide, promote and enhance the two highest-level goals, survival and reproduction.

In the case of artificial computational agents such as a web crawler or a robot or an autonomous unmanned vehicle, these agents are designed and programmed with a particular goal or a set of goals in designer's mind. Thus, the ability to act autonomously is typically related to an agent having some goal(s) to accomplish, and therefore being goal-driven.

From a systems perspective, in order for an agent to be reactive, it has to be *coupled* to its environment via some appropriate sensors (or "input channels") and effectors ("output channels"). Due to agent's sensors, the environment can affect the agent; due to agent's effectors, the agent can affect the outside environment. For a stronger notion of agency than *WAA*, in addition to some sort of sensors and effectors, necessary to ensure that the agent can interact with, affect and be affected by the outside world, it seems natural that an appropriate feedback, or control, loop exists between the agent and the outside world, so that this feedback loop affects how the agent responds to the environmental changes. A feedback loop provides the agent with knowledge of "how well it is doing". In particular, an agent will have use of a feedback loop only if it has an appropriate notion of its goals or tasks, and an evaluation function (task value, utility, re-

source consumption, or the like) associated with it. That is, an agent needs some sort of a *control loop* in order to be capable of goal-oriented or utility-oriented behavior.

Finally, in addition to responsiveness, persistence and goal-drivenness or goal-orientedness, one more characteristic found in nearly all interesting autonomous agents, not altogether unrelated to goal-orientedness, is that of *pro-activeness* [10, 18, 26]. While some literature on autonomous agents treats pro-activeness and goal-drivenness as synonyms, we briefly discuss why, in general, the two attributes ought to be distinguished.

Namely, a situated, reactive agent can be goal-oriented without being pro-active: given an input from the "world", the goal-oriented agent acts so as to ensure, e.g., avoiding being in certain of its internal states that it views incompatible with its limited knowledge of "what is going on out there". If there are no changes in the environment, the agent simply keeps "sitting" in whatever its current state happens to be. Thus, this reactive agent has a goal (although admittedly a very simplistic one), but is not pro-active. Similarly, an agent can be pro-active without being goal-oriented, as long as we require of agent's goal(s) to be non-trivial, and, in particular, to possibly entail some deliberate effect that the agent's actions may be required, under appropriate circumstances, to have on the environment. Under this assumption, an agent may "pro-actively" perform a more or less random walk among its internal states, without any observable effects on the outside world, and therefore without accomplishing - or, indeed, having - any specific goals insofar as the agent's deliberate influence on the environment.

Thus, while pro-activeness and goal-orientedness are usually closely related, they are not synonymous, and, moreover, neither subsumes the other.

Once a *WAA* agent is additionally equipped with some form of goal-drivenness and pro-activeness, we arrive at what we define as *strongly autonomous agency* (*SAA*). Most agents encountered in AI, whether they are software agents, robots, unmanned vehicles, or of any other kind, are of this, strongly autonomous type (see, e.g., [18, 16, 24]).

Therefore, we find that it is precisely the properties of (i) some degree of control of one's own internal state and behavior, (ii) reactivity or responsiveness, (iii) persistence, (iv) pro-activeness, and (v) goal-drivenness or goal-orientedness that, together, and in synergy with each other, make an agent *truly* (*or strongly*) *autonomous* in an AI sense:

$$\begin{aligned} \text{strong autonomous agency} &\approx \text{weak autonomous agency} \\ &+ \text{goal-orientedness} + \text{pro-activeness} \end{aligned}$$

Granted, much of the agent literature has identified properties (i) - (v) as common to autonomous agents (see, e.g., [26, 10] and references therein). We claim, however, that these five agent capabilities are *the necessary* properties that are all found in nearly every reasonable model of autonomous agency, whereas other characteristics, including sociability, mobility, "mental states", beliefs-desires-intentions, etc., are not as essential, and are found in (or can be reasonably attributed to) only *some*, but by no means

²This notion of weak agent autonomy, mainly based on the dominant notion of autonomous agents in the area of software design for open distributed systems, is obviously not called *weak* by those who consider it sufficient for their purposes.

(nearly) all of the known autonomous agents, whether biological or artificial.

However, even those living organisms that one would never consider intelligent have one more fundamental property, absent from our notion of SAA, and that is the ability to *adapt* (e.g., through metabolism). Adaptability is a necessary prerequisite for biological survival, as well as for any reasonable notion of intelligence. The “control loop” between the agent and the world serves no purpose, if the agent has no goals or notions of “goodness” with respect to which it tries to optimize its behavior. But such goal- or utility-drivenness is useless, if the agent cannot dynamically adjust its behavior based on the feedback, i.e., if it cannot adapt.

To summarize, based on how is an agent coupled to its environment, how complex properties of that environment the particular type of coupling (e.g., type of sensors, “control loop”, effectors) can capture, and how complex behaviors or actions the agent is capable of, we have proposed a tentative general hierarchical classification of autonomous agents embedded in, and acting as a part of, their environments.

Whether one would consider an agent that (i) has at least some control over its internal state and behavior, and is (ii) reactive, (iii) persistent, (iv) pro-active, (v) goal- or utility-driven, and (vi) adaptable, to automatically be *intelligent*, depends on one’s definition of intelligence and is subject to debate. What seems clear, however, is that no proper subset of the properties (i) - (vi) satisfies even the weakest notion of intelligence. Moreover, we argue that, as one keeps adding on properties from (i) towards (vi), one can recognize many well-known examples of agents found in the literature, yet not as a part of what we argue is a reasonable and natural hierarchy of agents³. For instance, artificial agents that possess only (i), (ii) and possibly (iii) are studied in detail in [17]. Some examples of WAA agents possessing (i) - (iii) and SAA agents having attributes (i) - (v) are discussed next.

5 Discussion and Some Applications

To illustrate the usefulness of the proposed hierarchy of agents, we consider some software engineering developments in the context of open distributed systems. Agent-oriented programming [20] can be viewed both as a novel paradigm and the natural successor to object-oriented paradigm [13]. The transition from object-oriented towards agent-oriented programming was motivated by the design of open distributed platforms, so that concurrency and resource sharing can be exploited in heterogeneous distributed environments [3].

To place the development of a general paradigm of autonomous agency into a broader computer science perspective, we briefly make a comparison to the development of the object-oriented paradigm. The primary motivation for moving away from function evaluation based classical imperative programming towards the object-oriented programming paradigm was primarily motivated by the nature of a

great number of emerging applications, where it was more natural to think in terms of objects and their classes and hierarchies, their capabilities (“methods”), etc., than in terms of functions being evaluated on variables. One particular domain that gave a huge impetus to the growth and success of object-oriented programming was that of computer simulation of various complex and distributed infrastructures [8]. As computing started becoming increasingly distributed both physically and logically, and these distributed systems getting increasingly heterogeneous, complex and open, the individual components, whether hardware or software, were moving away from non-autonomous components of a single, tightly coupled system, towards being increasingly sophisticated, autonomous and complex (sub)systems themselves, that were only loosely coupled into an overarching larger system. Hence, a novel paradigm capturing the increasing requirements in terms of autonomy, flexibility and complexity of the individual components in such distributed systems was sought - and, in case of the software, the *agent-based programming* paradigm was born [20].

We thus see the relationship of agent-oriented programming to object-oriented programming, in essence, similar to the relationship of object-oriented to classical imperative programming: each is a novel metaphor and a radical departure from its predecessor - yet a novel metaphor that clearly builds on the top of its predecessor, and adds more desirable properties that brings thus enhanced model considerably closer to the target applications.

Actors [1, 2] are a powerful model for specifying coordination in open distributed systems. In addition to its internal state, an actor also encapsulates its behavior (both data and procedure). An actor can communicate via asynchronous message passing with other actors; this asynchronous communication is, therefore, central to how actors interact with their environment. An actor is responsive or reactive; it also may (but need not) be persistent. Actors thus fit well into our concept of *weakly autonomous agency*.

Actors can also be used as a building block towards implementing more complex systems and, in particular, for software design of autonomous agents with stronger autonomous capabilities, via appropriate extensions (added functionality) of the basic actor model. For instance, ref. [3] addresses an important problem of how to extend actors into a powerful concurrent programming paradigm for *distributed artificial intelligence (DAI)* [6, 24]. There are other examples of designing strongly autonomous applications on the top of weakly autonomous infrastructures. For instance, an actor-based (hence, “weakly autonomous”) software infrastructure is used in [11, 12] to build a simulator of a particular kind of strongly autonomous agents, namely, autonomous unmanned vehicles [22, 23]. The basic agent capabilities are provided by the actor infrastructure, whereas the higher-order autonomous abilities, such as the pro-active pursuit of an agent’s goals or the agents’ coordination strategies, are built on the top of the basic actor architecture, i.e., at the “application software” level [11, 22, 23].

³However, see [10] for another proposal of a hierarchical taxonomy of various types of agents.

6 Conclusions

The subject of this paper are *autonomous agents* from a systems perspective. First, we survey some relevant literature and offer some general thoughts about various properties and notions of autonomous agents. We then propose a hierarchy of autonomous agents based on the complexity of their behaviors, and the necessary attributes that can yield particular behaviors. Instead of marking various types of agents as more or less complex in terms of the sophistication of their (supposed) mental or emotional states, we make distinction in terms of the basic agent capabilities whose presence - or lack thereof - is readily observable and measurable by an observer outside of the agent itself. Thus, instead of a “cognitive” or symbolic AI approach, we propose classifying autonomous agents in more behaviorist, functionalist and systems theory terms. In particular, we identify the three absolutely necessary properties for even the weak(est) notion of autonomous agency, and three additional, more advanced properties that are necessary for an agent, whether biological or artificial, to be reasonably considered deliberative or intelligent. We also show how some well-known, existing agent models fit into the appropriate layers of our proposed agent hierarchy. Finally, we point out some examples of how the lower-level agents can be used as “building blocks” in design of more complex, higher-level autonomous agents and *MAS*.

Acknowledgments: This work was supported by the DARPA IPTO TASK Program, contract F30602-00-2-0586.

References

- [1] G. Agha, “Actors: A Model of Concurrent Computation in Distributed Systems”, MIT Press, Cambridge, MA, 1986
- [2] G. Agha, “Concurrent Object-Oriented Programming”, in *Communications ACM*, vol. 33 (9), 1990
- [3] G. Agha, N. Jamali, “Concurrent Programming for DAI”, in G. Weiss (ed.), “Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence”, The MIT Press, 1999
- [4] W. Ross Ashby, “Design for a Brain”, Wiley, 1960
- [5] W. Ross Ashby, “An Introduction to Cybernetics”, 4th impression, Chapman & Hall Ltd., London, 1961
- [6] N. M. Avouris, L. Gasser (eds.), “Distributed Artificial Intelligence: Theory and Praxis”, Euro Courses Comp. & Info. Sci. vol. 5, Kluwer Academic Publ., 1992
- [7] M. E. Bratman, “Intentions, Plans and Practical Reason”, Harvard Univ. Press, Cambridge, MA, 1987
- [8] T. Budd, “An Introduction to Object-Oriented Programming”, Addison-Wesley, 1991
- [9] R. Dawkins, “The Selfish Gene”, Oxford Press, 1990
- [10] S. Franklin, A. Graesser, “Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents”, Proc. 3rd Int'l Workshop on Agent Theories, Architectures & Languages, Springer-Verlag, 1996
- [11] M. Jang, S. Reddy, P. Totic, L. Chen, G. Agha, “An Actor-based Simulation for Studying UAV Coordination”, Proc. 15th Euro. Symp. Simul. (ESS ’03), Delft, The Netherlands, October 2003
- [12] M. Jang, G. Agha, ”On Efficient Communication and Service Agent Discovery in Multi-agent Systems,” 3rd Int'l Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS ’04), Edinburgh, Scotland, May 2004
- [13] N. R. Jennings, “An agent-based approach for building complex software systems”, *Communications of the ACM* vol. 44 (4), 2001
- [14] G. J. Klir, “Facets of Systems Science” (2nd ed.), Plenum Press, New York, 2001
- [15] G. J. Klir, “Systems Science”, Encyclopedia of Information Systems, Vol. 4, Elsevier Sci. (USA), 2003
- [16] A. S. Rao, M. P. Georgeff, “BDI Agents: From Theory to Practice”, Proc. 1st Int'l Conf. on MAS (IC-MAS’95), San Francisco, USA, 1995
- [17] S. J. Rosenschein, L. P. Kaelbling, “A Situated View of Presentation and Control”, *Artificial Intelligence* vol. 73, 1995
- [18] S. Russell, P. Norvig, “Artificial Intelligence: A Modern Approach”, 2nd ed., Prentice Hall, 2003
- [19] John Searle, “Minds, Brains and Science”, Harvard University Press, Cambridge, 1984
- [20] Y. Shoham, “Agent-oriented programming”, *Artificial Intelligence* vol. 60, 1993
- [21] H. A. Simon, “Models of Man”, J. Wiley & Sons, New York, 1957
- [22] P. Totic, M. Jang, S. Reddy, J. Chia, L. Chen, G. Agha, “Modeling a System of UAVs on a Mission”, Proc. SCI’03, Orlando, Florida, July 2003
- [23] P. Totic, G. Agha, “Modeling Agent’s Autonomous Decision Making in Multi-Agent, Multi-Task Environments”, Proc. 1st Euro. Workshop on MAS (EU-MAS’03), Oxford, England, December 2003
- [24] G. Weiss (ed.), “Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence”, The MIT Press, Cambridge, MA, 1999
- [25] N. Wiener, “Cybernetics”, J. Wiley, New York, 1948
- [26] M. Wooldridge, N. Jennings, “Intelligent Agents: Theory and Practice”, Knowledge Engin. Rev., 1995

Characterizing Configuration Spaces of Simple Threshold Cellular Automata

Predrag T. Tomic and Gul A. Agha

Open Systems Laboratory, Department of Computer Science

University of Illinois at Urbana-Champaign

Mailing address: Siebel Center for Computer Science,

201 N. Goodwin Ave., Urbana, IL 61801, USA

p-tomic@cs.uiuc.edu, agha@cs.uiuc.edu

Abstract. We study herewith *the simple threshold cellular automata (CA)*, as perhaps the simplest broad class of *CA* with non-additive (i.e., non-linear and non-affine) local update rules. We characterize all possible computations of the most interesting rule for such *CA*, namely, the *Majority (MAJ)* rule, both in the classical, parallel *CA* case, and in case of the corresponding sequential *CA* where the nodes update sequentially, one at a time. We compare and contrast the configuration spaces of arbitrary simple threshold automata in those two cases, and point out that some parallel threshold *CA* cannot be simulated by any of their sequential counterparts. We show that the temporal cycles exist only in case of (some) parallel simple threshold *CA*, but can never take place in sequential threshold *CA*. We also show that most threshold *CA* have very few fixed point configurations and few (if any) cycle configurations, and that, while the *MAJ* sequential and parallel *CA* may have many fixed points, nonetheless “almost all” configurations, in both parallel and sequential cases, are transient states.

1 Introduction and Motivation

Cellular automata (CA) were originally introduced as an abstract mathematical model of the behavior of biological systems capable of self-reproduction [15]. Subsequently, variants of *CA* have been extensively studied in a great variety of application domains, predominantly in the context of complex physical or biological systems and their dynamics (e.g., [20, 21, 22]). However, *CA* can also be viewed as an abstraction of massively parallel computers (e.g., [7]). Herein, we study a particular simple yet nontrivial class of *CA* from a computer science perspective. This class are the *threshold cellular automata*. In the context of such *CA*, we shall first compare and contrast the configuration spaces of the classical, concurrent *CA* and their sequential analogues. We will then pick a particular threshold node update rule, and fully characterize possible computations in both parallel and sequential cases for the one-dimensional automata.

Cellular automata CA are an abstract computational model of *fine-grain parallelism* [7], in that the elementary operations executed at each node are rather simple and hence comparable to the basic operations performed by the computer hardware. In a classical, that is, concurrently executing *CA*, whether finite or infinite, all the nodes execute their operations *logically simultaneously*: the state of a node x_i at time step

$t + 1$ is some simple function of the states (i) of the node x_i itself, and (ii) of a set of its pre-specified neighbors, at time t .

We consider herewith the sequential version of CA, heretofore abridged to SCA, and compare such sequential CA with the classical, *parallel (concurrent)* CA. In particular, we show that there are 1-D CA with very simple node state update rules that cannot be simulated by any comparable SCA, irrespective of the node update ordering.

We also fully characterize the possible computations of the most interesting case of *threshold cellular automata*, namely, the (S)CA with the *Majority* node update rule.

An important remark is that we use the terms *parallel* and *concurrent* as synonyms throughout the paper. This is perhaps not the most standard convention, but we are not alone in not making the distinction between the two terms (cf. discussion in [16]). Moreover, by a *parallel* (equivalently, *concurrent*) *computation* we shall mean actions of several processing units that are carried out *logically* (if not necessarily *physically*) *simultaneously*. In particular, when referring to *parallel* or *concurrent* computation, we do assume a *perfect synchrony*.

2 Cellular Automata and Types of Their Configurations

We follow [7] and define classical (that is, synchronous and concurrent) CA in two steps: by first defining the notion of a *cellular space*, and subsequently that of a *cellular automaton* defined over an appropriate cellular space.

Definition 1: A *Cellular Space*, Γ , is an ordered pair (G, Q) where G is a regular graph (finite or infinite), with each node labeled with a distinct integer, and Q is a finite set of states that has at least two elements, one of which being the special *quiescent state*, denoted by 0.

We denote the set of integer labels of the nodes (vertices) in Γ by L .

Definition 2: A *Cellular Automaton* (CA), \mathbf{A} , is an ordered triple (Γ, N, M) where Γ is a *cellular space*, N is a *fundamental neighborhood*, and M is a *finite state machine* such that the input alphabet of M is $Q^{|N|}$, and the local transition function (update rule) for each node is of the form $\delta : Q^{|N|+1} \rightarrow Q$ for CA with *memory*, and $\delta : Q^{|N|} \rightarrow Q$ for *memoryless CA*.

Some of our results pertain to a comparison and contrast between the classical, concurrent threshold CA and their sequential counterparts, the threshold SCA.

Definition 3: A *Sequential Cellular Automaton* (SCA) \mathbf{S} is an ordered quadruple (Γ, N, M, s) , where Γ, N and M are as in Def. 2, and s is a sequence, finite or infinite, all of whose elements are drawn from the set L of integers used in labeling the vertices of Γ . The sequence s is specifying the sequential ordering according to which an SCA's nodes update their states, one at a time.

However, when comparing and contrasting the concurrent threshold CA with their sequential counterparts, rather than making a comparison between a given CA with a *particular SCA*, we compare the parallel CA computations with the computations of the corresponding SCA for *all* possible sequences of node updates. To that end, the following convenient terminology is introduced:

Definition 4: A *Nondeterministic Interleavings Cellular Automaton* (NICA) \mathbf{I} is defined to be the union of all sequential automata \mathbf{S} whose first three components, Γ, N

and M , are fixed. That is, $\mathbf{I} = \bigcup_s (\Gamma, N, M, s)$, where the meanings of Γ, N, M , and s are the same as before, and the union is taken over all (finite and infinite) sequences $s : \{1, 2, 3, \dots\} \rightarrow L$ (where L is the set of integer labels of the nodes in Γ).

Since our goal is to characterize all possible computations of parallel and sequential threshold CA, a (*discrete*) *dynamical system* view of CA will be useful. A *phase space* of a dynamical system is a (finite or infinite, as appropriate) directed graph where the vertices are the *global configurations* (or *global states*) of the system, and directed edges correspond to possible transitions from one global state to another. We now define the fundamental, qualitatively distinct types of (global) configurations that a classical (parallel) cellular automaton can find itself in.

Definition 5: A *fixed point* (FP) is a configuration in the phase space of a CA such that, once the CA reaches this configuration, it stays there forever. A (*proper*) *cycle configuration* (CC) is a state that, if once reached, will be revisited infinitely often with a fixed, finite period of 2 or greater. A *transient configuration* (TC) is a state that, once reached, is never going to be revisited again.

In particular, FPs are a special, degenerate case of recurrent states whose period is 1. Due to their deterministic evolution, any configuration of a classical, parallel CA belongs to exactly one of these basic configuration types, i.e., it is a FP, a proper CC, or a TC. On the other hand, if one considers *sequential CA* so that *arbitrary* node update orderings are permitted, that is, if one considers *NICA* automata, then, given the underlying cellular space and the local update rule, the resulting phase space configurations, due to nondeterminism that results from different choices of possible sequences of node updates, are more complicated. In a particular SCA, a cycle configuration is any configuration revisited infinitely often - but the period between different consecutive visits, assuming an arbitrary sequence s of node updates, need not be fixed. We call a global configuration that is revisited only finitely many times (under a given ordering s) *quasi-cyclic*. Similarly, a *quasi-fixed point* is a SCA configuration such that, once the dynamics reaches this configuration, it stays there “for a while” (i.e., for some finite number of sequential node update steps), and then leaves. For example, a configuration of a SCA can be simultaneously a (quasi-)FP and a (quasi-)CC (see, e.g., the example in [19]). For simplicity, heretofore we shall refer to a configuration x of a *NICA* as a *pseudo fixed point* if there exists some infinite sequence of node updates s such that x is a FP in the usual sense when the corresponding SCA’s nodes update according to the ordering s . A global configuration of a *NICA* is a *proper* FP iff it is a fixed point of each corresponding SCA, that is, for every sequence of node updates s . Similarly, we consider a global configuration y of a *NICA* to be a cycle state, if there exists an infinite sequence of the node updates s' such that, if the corresponding SCA’s nodes update according to s' , then y is a recurrent state and, moreover, y is not a *proper* FP. Thus, in general, a global configuration of a *NICA* automaton can be simultaneously a (pseudo) FP, a CC and a TC (with respect to different node update sequences s)¹.

¹ When the allowable sequences of node updates $s : \{1, 2, 3, \dots\} \rightarrow L$ are required to be infinite and *fair* so that, in particular, every (infinite) tail $s^{[n]} : \{n, n+1, n+2, \dots\} \rightarrow L$ is *onto* L , then *pseudo fixed points* and *proper fixed points* in *NICA* can be shown to coincide with one another and, moreover, with the “ordinary” FPs for parallel CA. For the special case when L is finite and s is required to be an *ad infinitum* repeated permutation see, e.g., [3, 4].

Definition 6: A *1-D cellular automaton of radius r* ($r \geq 1$) is a *CA* defined over a one-dimensional string of nodes, such that each node's next state depends on the current states of its neighbors to the left and to the right that are no more than r nodes away (and, in case of the *CA with memory*, on the current state of that node itself).

We adopt the following conventions and terminology. Throughout, only *Boolean CA* and *SCA/NICA* are considered; in particular, the set of possible states of any node is $\{0, 1\}$. The terms “monotone symmetric” and “symmetric (linear) threshold” functions/update rules/automata are used interchangeably. Similarly, the terms “(global) dynamics” and “(global) computation” are used synonymously. Also, unless explicitly stated otherwise, automata *with memory* are assumed. The default *infinite* cellular space Γ is a two-way infinite line. The default *finite* Γ is a ring with an appropriate number of nodes². The terms “phase space” and “configuration space” will be used synonymously, as well, and sometimes abridged to *PS*.

3 Properties of 1-D Simple Boolean Threshold CA and SCA

Herein, we compare and contrast the classical, parallel *CA* with their sequential counterparts, *SCA* and *NICA*, in the context of the simplest (nonlinear) local update rules possible, namely, the *Boolean linear threshold rules*. Moreover, we choose these threshold functions to be *symmetric*, so that the resulting *CA* are also *totalistic* (see, e.g., [7] or [21]). We show the fundamental difference in the configuration spaces, and therefore possible computations, in case of the classical, concurrent threshold automata on one, and the sequential threshold cellular automata, on the other hand: while the former can have temporal cycles (of length two), the computations of the latter either do not converge at all after any finite number of sequential steps, or, if the convergence does take place, it is *necessarily* to a fixed point.

First, we need to define *threshold functions*, *simple threshold functions*, and the corresponding types of *(S)CA*.

Definition 7: A *Boolean-valued linear threshold function* of n inputs, x_1, \dots, x_n , is any function of the form

$$f(x_1, \dots, x_n) = \begin{cases} 1, & \text{if } \sum_i w_i \cdot x_i \geq \theta \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where θ is an appropriate *threshold constant*, and w_i are real-valued *weights*.

Definition 8: A *threshold cellular automaton* is a (parallel or sequential) cellular automaton such that its node update rule δ is a *Boolean-valued linear threshold function*.

² It turns out, that circular boundary conditions are important for some of our technical results. Likewise, some results about the phase space properties of concurrent and sequential threshold *CA* may require (i) a certain minimal number of nodes and (ii) that the number of nodes be, e.g., even, divisible by four, or the like. Heretofore, we shall assume a sufficient number of nodes that “works” in the particular situation, without detailed elaborations.

Definition 9: A *simple* threshold $(S)CA$ is an automaton whose local update rule δ is a monotone symmetric Boolean (threshold) function.

Throughout, whenever we say *a threshold automaton (threshold CA)*, we shall mean *simple threshold automaton (threshold CA)* - unless explicitly stated otherwise.

Due to the nature of the node update rules, cyclic behavior intuitively should not be expected in these simple threshold automata. This is, generally, (almost) the case, as will be shown below. We argue that the importance of the results in this section largely stems from the following three factors: (i) the local update rules are the simplest nonlinear totalistic rules one can think of; (ii) given the rules, the cycles are not to be expected - yet they exist, and in the case of classical, parallel *CA only*; and, related to that observation, (iii) it is, for this class of $(S)CA$, the parallel *CA* that exhibit the more interesting behavior than any corresponding sequential *SCA* (and consequently also *NICA*) [19], and, in particular, while there is nothing (qualitatively) among the possible sequential computations that is not present in the parallel case, the classical parallel threshold *CA* are capable of a particular qualitative behavior - namely, they may have nontrivial temporal cycles - that cannot be reproduced by *any* simple threshold *SCA* (and, therefore, also threshold *NICA*).

The results below hold for the two-way infinite *1-D CA*, as well as for the finite *CA* and *SCA* with sufficiently many nodes and circular boundary conditions.

Lemma 1: (i) A 1-D classical (i.e., parallel) *CA* with $r = 1$ and the *Majority* update rule has (finite) temporal cycles in the phase space (PS). In contrast, (ii) *1-D Sequential CA* with $r = 1$ and the *Majority* update rule do not have any (finite) cycles in the phase space, *irrespective* of the sequential node update order s . \diamond

Remarks: In case of infinite sequential *SCA* as in the *Lemma* above, a nontrivial cycle configuration does not exist even in the limit. In finite cases, s is an arbitrary sequence of an *SCA* nodes' indices, not necessarily a (repeated) permutation.

We thus conclude that *NICA* with $\delta = MAJ$ and $r = 1$ are temporal cycle-free. Moreover, it turns out that, even if we consider local update rules δ other than the *MAJ* rule, yet restrict δ to *monotone symmetric Boolean functions*, such sequential *CA* still do not have any temporal cycles.

Lemma 2: For any *Monotone Symmetric Boolean 1-D Sequential CA* S with $r = 1$, and any sequential update order s , the phase space $PS(S)$ is cycle-free. \diamond

Similar results to those in *Lemmata 1-2* also hold for *1-D CA* with radius $r \geq 2$.

Theorem 1: (i) *1-D (parallel) CA* with $r \geq 1$ and with the *Majority* node update rule have (finite) cycles in the phase space. (ii) Any *1-D SCA* with $\delta = MAJ$ or any other monotone symmetric Boolean node update rule, $r \geq 1$ and any sequential order s of the node updates has a cycle-free phase space. \diamond

Remarks: The claims of *Thm. 1* hold both for the finite $(S)CA$ (provided that they have sufficiently many nodes, an even number of nodes in case of the *CA* with cycles, and assuming the circular boundary conditions in *part (i)*), and for the infinite $(S)CA$. We also observe that several variants of the result in *Theorem 1 (ii)* can be found in the literature. When the sequence of node updates of a finite *SCA* is periodic, with a single period a fixed permutation of the nodes, the temporal cycle-freeness of sequential *CA* and many other properties can be found in [8] and references therein. In [4], fixed

permutation of the sequential node updates is also required, but the underlying cellular space Γ is allowed to be an arbitrary finite graph, and different nodes are allowed to compute *different* simple k -threshold functions.

As an immediate consequence of the results presented thus far, we have

Corollary 1: For all $r \geq 1$, there exists a *monotone symmetric CA* (that is, a *threshold automaton*) \mathbf{A} such that \mathbf{A} has finite temporal cycles in the phase space.

Some of the results for $(S)CA$ with $\delta = MAJ$ do extend to some, but by no means all, other simple threshold $(S)CA$ defined over the same cellular spaces. For instance, consider the k -threshold functions with $r = 2$. There are five nontrivial such functions, for $k \in \{1, 2, 3, 4, 5\}$. The 1-threshold function is Boolean *OR* function (in this case, on $2r + 1 = 5$ inputs), and the corresponding *CA* do not have temporal cycles; likewise with the “5-threshold” *CA*, that update according to Boolean *AND* on five inputs. However, in addition to *Majority* (i.e., 3-threshold), it is easy to show that 2-threshold (and therefore, by symmetry, also 4-threshold) such *CA* with $r = 2$ do have temporal two-cycles; for example, in the 2-threshold case, for *CA* defined over an infinite line, $\{(1000)^\omega, (0010)^\omega\}$ is a two-cycle.

We now relate our results thus far to what has been already known about simple threshold *CA* and their phase space properties. In particular, the only recurrent types of configurations we have identified thus far are FPs (in the sequential case), and FPs and two-cycles, in the concurrent *CA* case. This is not a coincidence.

It turns out that the two-cycles in the *PS* of the parallel *CA* with $\delta = MAJ$ are actually the only type of (proper) temporal cycles such cellular automata can have. Indeed, for any *symmetric linear threshold update rule* δ , and any *finite* regular Cayley graph as the underlying cellular space, the following general result holds (see [7, 8]):

Proposition 1: Let a classical *CA* $\mathbf{A} = (\Gamma, N, T)$ be such that Γ is finite and the underlying local rule of T is an elementary symmetric threshold function. Then for all configurations $C \in PS(\mathbf{A})$, there exists $t \geq 0$ such that $T^{t+2}(C) = T^t(C)$. \diamond

In particular, this result implies that, in case of any finite simple threshold automaton, and for any starting configuration C_0 , there are only two possible kinds of orbits: upon repeated iteration, after finitely many steps, the computation either converges to a fixed point configuration, or else it converges to a two-cycle³.

We now specifically focus on $\delta = MAJ$ 1-D *CA*, with an emphasis on the infinite case, and completely characterize the configuration spaces of such threshold automata. In particular, in the $\Gamma = \text{infinite line}$ case, we show that the cycle configurations are rather rare, that fixed point configurations are quite numerous - yet still relatively rare in a sense to be discussed below, and that *almost all* configurations of these threshold automata are transient states.

Henceforth, insofar as the *SCA* and *NICA* automata were concerned, for the most part we have allowed entirely *arbitrary* sequences s of node updates, or at least *arbitrary infinite* such sequences. In order to carry the results on FPs and TCs of (parallel) *MAJ CA* over to the sequential automata with $\delta = MAJ$ (and, when applicable, other

³ If one considers *threshold (S)CA* defined over infinite Γ , the only additional possibility is that such automaton’s dynamic evolution fails to converge after any finite number of steps.

simple threshold rules) as well, throughout the rest of the paper we will allow *fair sequences only*: that is, we shall now consider only those threshold SCA (and NICA) where each node gets its turn to update *infinitely often*. In particular, this ensures that (i) any pseudo FP of a given NICA is also a proper FP, and (ii) the FPs of a given parallel CA coincide with the (proper) FPs of the corresponding SCA and NICA.

We begin with some simple observations about the nature of various configurations in the $(S)CA$ with $\delta = MAJ$ and $r = 1$. We shall subsequently generalize most of these results to arbitrary $r \geq 1$. We first recall that, for such $(S)CA$ with $r = 1$, two adjacent nodes of the same value are stable. That is, 11 and 00 are stable sub-configurations. Consider now the starting sub-configuration $x_{i-1}x_i x_{i+1} = 101$. In the parallel case, at the next time step, $x_i \rightarrow 1$. Hence, no FP configuration of a parallel CA can contain 101 as a sub-configuration. In the sequential case, assuming fairness, x_i will eventually have to update. If, at that time, it is still the case that $x_{i-1} = x_{i+1} = 1$, then $x_i \rightarrow 1$, and $x_{i-1}x_i x_{i+1} \rightarrow 111$, which is stable. Else, at least one of x_{i-1}, x_{i+1} has already “flipped” into 0. Without loss of generality, let’s assume $x_{i-1} = 0$. Then $x_{i-1}x_i = 00$, which is stable; so, in particular, $x_{i-1}x_i x_{i+1}$ will never go back to the original 101. By symmetry of $\delta = MAJ$ with respect to 0 and 1, the same line of reasoning applies to the sub-configuration $x_{i-1}x_i x_{i+1} = 010$. In particular, the following properties hold:

Lemma 3: A fixed point configuration of a 1D-(S)CA with $\delta = Majority$ and $r = 1$ cannot contain sub-configurations 101 or 010. Similarly, a cycle configuration of such a 1D-(S)CA cannot contain sub-configurations 00 or 11. \diamond

Of course, we have already known that, in the sequential case, no cycle states exist, period. In case of the parallel threshold CA, by virtue of determinism, a complete characterization of each of the three basic types of configurations (FPs, CCs, TCs) is now almost immediate:

Lemma 4: The FPs of the 1D-(S)CA with $\delta = MAJ$ and $r = 1$ are precisely of the form $(000^* + 111^*)^*$. The CCs of such 1D-CA exist only in the concurrent case, and the temporal cycles are precisely of the form $\{(10)^*, (01)^*\}$. All other configurations are *transient states*, that is, TCs are precisely the configurations that contain both (i) 000^* or 111^* (or both), and (ii) 101 or 010 (or both) as their sub-configurations. In addition, the CCs in the parallel case become TCs in all corresponding sequential cases. \diamond

Some generalizations to arbitrary (finite) rule radii r are now immediate. For instance, given any such $r \geq 1$, the finite sub-configurations 0^{r+1} and 1^{r+1} are stable with respect to $\delta = MAJ$ update rule applied either in parallel or sequentially; consequently, any configuration of the form $(0^{r+1}0^* + 1^{r+1}1^*)^*$, for both finite and infinite $(S)CA$, is a fixed point. This characterization, only with a considerably different notation, has been known for the case of configurations with *compact support* for a relatively long time; see, e.g., *Chapter 4* in [8]. On the other hand, fully characterizing CCs (and, consequently, also TCs) in case of finite or infinite (parallel) CA is more complicated than in the simplest case with $r = 1$. For example, for $r \geq 1$ odd, and $\Gamma = infinite\ line$, $\{(10)^\omega, (01)^\omega\}$ is a two-cycle, whereas for $r \geq 2$ even, each of $(10)^\omega, (01)^\omega$ is a fixed point. However, for all $r \geq 1$, the corresponding (parallel) CA are guaranteed to have some temporal cycles, namely, given $r \geq 1$, the doubleton of states $\{(1^r 0^r)^\omega, (0^r 1^r)^\omega\}$ forms a temporal two-cycle.

Lemma 5: Given any (finite or infinite) threshold $(S)CA$, one of the following two properties always holds: either (i) this threshold automaton does not have proper cycles and cycle states; or (ii) if there are cycle states in the PS of this automaton, then none of those cycle states has any incoming transients. \diamond

Moreover, if there are any (two-)cycles, the number of these temporal cycles and therefore of the cycle states is, statistically speaking, negligible:

Lemma 6: Given an infinite $MAJ CA$ and a finite radius of the node update rules $r \geq 1$, among uncountably many (2^{\aleph_0} , to be precise) global configurations of such a CA , there are only finitely many (proper) cycle states. \diamond

On the other hand, fixed points of some threshold automata are *much more numerous* than the CCs. The most striking are the $MAJ(S)CA$ with their abundance of FPs. Namely, the cardinality of the set of FPs, in case of $\delta = MAJ$ and (countably) infinite cellular spaces, equals the cardinality of the entire PS :

Theorem 2: An infinite 1D- $(S)CA$ with $\delta = MAJ$ and any $r \geq 1$ has *uncountably many* fixed points. \diamond

The above result is another evidence that “not all threshold $(S)CA$ are born equal”. It suffices to consider only 1D, infinite CA to see a rather dramatic difference. Namely, in contrast to the $\delta = MAJ CA$, the CA with memory and with $\delta \in \{OR, AND\}$ (i) do not have any temporal cycles, and (ii) have *exactly two* FPs, namely, 0^ω and 1^ω . Other threshold CA may have temporal cycles, as we have already shown, but they still have only a finite number of FPs.

We have just argued that 1-D infinite $MAJ(S)CA$ have uncountably many FPs. However, these FPs are, when compared to the transient states, still but a few. To see this, let’s assume that a “random” global configuration is obtained by “picking” each site’s value to be either 0 or 1 at random, with equal probability, and so that assigning a value to one site is independent of the value assignment to any of the other sites. Then the following result holds:

Lemma 7: If a global configuration of an infinite threshold automaton is selected “at random”, that is, by assigning each node’s value independently and according to a toss of a fair coin, then, with probability 1, this randomly chosen configuration will be a transient state. \diamond

Moreover, the “unbiased randomness”, while sufficient, is certainly not necessary. In particular, assigning bit values according to outcomes of tossing a coin with a fixed bias also yields transient states being of probability one.

Theorem 3: Let p be any real number such that $0 < p < 1$, and let the probability of a site in a global configuration of a threshold automaton being in state 1 be equal to p (so that the probability of this site’s state being 0 is equal to $q = 1 - p$). If a global configuration of this threshold automaton is selected “at random” where the state of each node is an *i.i.d. discrete random variable* according to the probability distribution specified by p , then, with probability 1, this global configuration will be a transient state. \diamond

In case of the finite threshold $(S)CA$, as the number of nodes, N , grows, the fraction of the total of 2^N global configurations that are TCs will also tend to grow.

In particular, under the same assumptions as above, in the limit, as $N \rightarrow \infty$, the probability that a randomly picked configuration, C , is a transient state approaches 1:

$$\lim_{N \rightarrow \infty} \Pr(C \text{ is transient}) = 1 \quad (2)$$

Thus, a fairly complete characterization of the configuration spaces of threshold CA/SCA/NICA over finite and infinite 1-D cellular spaces can be given. In particular, under a simple and reasonable definition of what is meant by a “randomly chosen” global configuration in the infinite threshold CA case, *almost every* configuration of such a CA is a TC. However, when it comes to the number of fixed points, the striking contrast between $\delta = \text{MAJ}$ and all other threshold rules remains: in the infinite Γ cases, the MAJ CA have uncountably many FPs, whereas all other simple threshold CA have only finitely many FPs. The same characterizations hold for the *proper* FPs of the corresponding *simple threshold NICA* automata.

4 Conclusion

The theme of this work is a study of the fundamental configuration space properties of *simple threshold cellular automata*, both when the nodes update synchronously in parallel, and when they update sequentially, one at a time.

Motivated by the well-known notion of the sequential interleaving semantics of concurrency, we apply the “*interleaving semantics*” metaphor to the parallel CA and thus motivate the study of sequential cellular automata, SCA and NICA, and the comparison and contrast between SCA and NICA on one, and the classical, concurrent CA, on the other hand [19]. We have shown that even in this simplistic context, the perfect synchrony of the classical CA node updates has some important implications, and that the sequential CA cannot capture certain aspects of their parallel counterparts’ behavior. Hence, simple as they may be, the basic operations (local node updates) in classical CA cannot always be considered atomic. Thus we find it reasonable to consider a single local node update to be made of an ordered sequence of finer elementary operations: (i) fetching (“receiving”?) all the neighbors’ values, (ii) updating one’s own state according to the update rule δ , and (iii) making available (“sending”?) one’s new state to the neighbors.

We also study in some detail perhaps the most interesting of all simple threshold rules, namely, the *Majority* rule. In particular, we characterize all three fundamental types of configurations (transient states, cycle states and fixed point states) in case of finite and infinite 1D-CA with $\delta = \text{MAJ}$ for various finite rule radii $r \geq 1$. We show that CCs are, indeed, a rare exception in such MAJ CA, and that, for instance, the infinite MAJ (S)CA have uncountably many FPs, in a huge contrast to other simple threshold rules that have only a handful of FPs. We also show that, assuming a random configuration is chosen via independently assigning to each node its state value by tossing a (not necessarily fair) coin, it is very likely, for a sufficiently large number of the automaton’s nodes, that this randomly chosen configuration is a TC.

To summarize, the class of the simple threshold CA, SCA, and NICA is (i) relatively broad and interesting, and (ii) nonlinear (non-additive), yet (iii) all of these automata’s long-term behavior patterns can be readily characterized and effectively predicted.

Acknowledgments: The work presented herein was supported by the *DARPA IPTO TASK Program*, contract number *F30602-00-2-0586*.

References

1. W. Ross Ashby, “Design for a Brain”, Wiley, 1960
2. C. Barrett and C. Reidys, “Elements of a theory of computer simulation I: sequential CA over random graphs”, Applied Math. & Comput., vol. 98 (2-3), 1999
3. C. Barrett, H. Hunt, M. Marathe, S. S. Ravi, D. Rosenkrantz, R. Stearns, and P. Tocino, “Gardens of Eden and Fixed Points in Sequential Dynamical Systems”, Discrete Math. & Theoretical Comp. Sci. Proc. AA (DM-CCG), July 2001
4. C. Barrett, H. B. Hunt III, M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, R. E. Stearns, “Reachability problems for sequential dynamical systems with threshold functions”, TCS 1-3: 41-64, 2003
5. C. Barrett, H. Mortveit, and C. Reidys, “Elements of a theory of computer simulation II: sequential dynamical systems”, Applied Math. & Comput. vol. 107(2-3), 2000
6. C. Barrett, H. Mortveit, and C. Reidys, “Elements of a theory of computer simulation III: equivalence of sequential dynamical systems”, Appl. Math. & Comput. vol. 122(3), 2001
7. Max Garzon, “Models of Massive Parallelism: Analysis of Cellular Automata and Neural Networks”, Springer, 1995
8. E. Goles, S. Martinez, “Neural and Automata Networks: Dynamical Behavior and Applications”, Math. & Its Applications series (vol. 58), Kluwer, 1990
9. E. Goles, S. Martinez (eds.), “Cellular Automata and Complex Systems”, Nonlinear Phenomena and Complex Systems series, Kluwer, 1999
10. T. E. Ingerson and R. L. Buvel, “Structure in asynchronous cellular automata”, Physica D: Nonlinear Phenomena, vol. 10 (1-2), Jan. 1984
11. S. A. Kauffman, “Emergent properties in random complex automata”, Physica D: Nonlinear Phenomena, vol. 10 (1-2), Jan. 1984
12. Robin Milner, “A Calculus of Communicating Systems”, Lecture Notes Comp. Sci., Springer, Berlin, 1989
13. Robin Milner, “Calculi for synchrony and asynchrony”, Theoretical Comp. Sci. 25, Elsevier, 1983
14. Robin Milner, “Communication and Concurrency”, C. A. R. Hoare series ed., Prentice-Hall Int’l, 1989
15. John von Neumann, “Theory of Self-Reproducing Automata”, edited and completed by A. W. Burks, Univ. of Illinois Press, Urbana, 1966
16. J. C. Reynolds, “Theories of Programming Languages”, Cambridge Univ. Press, 1998
17. Ravi Sethi, “Programming Languages: Concepts & Constructs”, 2nd ed., Addison-Wesley, 1996
18. K. Sutner, “Computation theory of cellular automata”, MFCS98 Satellite Workshop on CA, Brno, Czech Rep., 1998
19. P. Tocino, G. Agha, “Concurrency vs. Sequential Interleavings in 1-D Cellular Automata”, APDCM Workshop, Proc. IEEE IPDPS’04, Santa Fe, New Mexico, 2004
20. Stephen Wolfram “Twenty problems in the theory of CA”, Physica Scripta 9, 1985
21. Stephen Wolfram (ed.), “Theory and applications of CA”, World Scientific, Singapore, 1986
22. Stephen Wolfram, “Cellular Automata and Complexity (collected papers)”, Addison-Wesley, 1994
23. Stephen Wolfram, “A New Kind of Science”, Wolfram Media, Inc., 2002

This article was processed using the **LATEX** macro package with LLNCS style

ON CHALLENGES IN MODELING AND DESIGNING RESOURCE-BOUNDED AUTONOMOUS AGENTS ACTING IN COMPLEX DYNAMIC ENVIRONMENTS

Predrag T. Totic and Gul A. Agha

Open Systems Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign
1334 Siebel Center for Computer Science, 201 N. Goodwin, Urbana, IL 61801, USA
Email: {p-totic, agha}@cs.uiuc.edu

ABSTRACT

Applications in which agent ensembles reside and operate in dynamically changing environments and where the agents are required to accomplish various tasks pose a number of modeling, design and analysis challenges. A necessary prerequisite for effective design and analysis of deliberative autonomous agents acting in complex dynamic environments is to have a good model of agents' environments and goals. In particular, a thorough theory of *Multi-Agent Systems (MAS)* requires modeling the interaction and coupling among an agent's behavior, properties of the environment, and the nature of an agent's goals in terms of the desired and/or achieved effects of the agent's behavior on the environment. This paper focuses on some of the challenges of understanding and modeling the relationship between an agent and its behavior on one, and the nature of the agent's environment and tasks, on the other hand. We also discuss the promises and limitations of the classical decision theory when applied to an individual agent's autonomous decision making in such *MAS* scenarios where the measure of an agent's successfulness is a function of the critical parameters of the agent's environment, and where this environment is dynamic, complex and partially inaccessible to the agent.

KEYWORDS

multi-agent systems, autonomous agents, resource-bounded agents, bounded rationality, decision theory

1 Introduction

Autonomous agents are a growing and increasingly exciting research area in many scientific disciplines, from economics to social sciences to software engineering to artificial intelligence (e.g., [1, 22, 23]). However, there is no unique, broadly agreed upon notion of agency in general, and autonomous agency in particular [4]. During the early stages of the area of *multi-agent systems (MAS)* and *distributed artificial intelligence (DAI)*, the default assumption was that different agents were designed and deployed by the same user, and, consequently, that all these different agents shared the same goal. That is, under this assumption, the main problem of *DAI* is the problem of *distributed control*. More generally, classical *DAI* is chiefly about *distributed problem solving (DPS)* (e.g., [1]). Thus, the agents employed in distributed control type problems, where the design goals include optimal allocation

of subtasks and resources, and where optimality or effectiveness are defined with respect to the system as a whole, are typically not *self-interested*. On the other hand, the agents that are self-interested, and have their own agendas and goals tend, in general, to be more complex to both design and analyze - and, in particular, can be expected to require a higher degree of individual autonomy. For our purposes, the main characteristic of this individual autonomy is an agent's capability of goal-driven (or utility-driven) autonomous decision making.

We shall consider herein a simplified version of an agent's decision making problem, that of an *action selection*: among a finite set of available actions, an agent is to select the one it finds the best or most optimal, given some notion of optimality, a measure of "goodness", and (limitations of) the agent's knowledge about the state of the world.

We are primarily interested in *self-interested agents* that, nonetheless, may need to coordinate with one another - *including cooperative or collaborative forms of coordination*. There are also those *MAS* applications where, from the designer's perspective, individual agents need not be self-interested in the strict sense, and where there exists an overall shared set of *goals* at the system level. That is, there is an overall objective that all agents share - yet maintaining the knowledge of these goals in a dynamic, distributed and, in many applications, hard real-time setting, may not be feasible. In such situations, it seems reasonable to assume that the agents are indeed self-interested. Each agent is thus assumed only to have its individual goals, and such an agent only needs to maintain the knowledge of its own goals, but need not worry about maintaining any shared knowledge about the goals or the state of the environment with other agents. For the large-scale multi-agent systems of this sort, the classical distributed control approach would likely not scale well. Therefore, an alternative is to "endow" the agents with higher-level capabilities of *individual* autonomous decision making. The *MAS* designer then, initially, "pretends" that each agent solely has its own agenda, without having any shared goals with other agents. Once scalability and robustness of the system have been achieved (see, e.g., [5]), the system designer can then focus on how to translate good or optimal individual behaviors (with respect to the individual utility functions) into a good or optimal system behavior (with respect to an imposed by the designer global utility). This, essentially, is the problem of *incentive engineering* [3]. We shall not specifically address this issue herein due to space constraints.

An example of a class of application domains such that, in order to achieve system scalability, robustness, dependability and other desired properties, a high degree of agent autonomy may be required, are the systems of autonomous, unmanned vehicles that are operating in complex, dynamic and partially observable environments. These agents have to accomplish some task or a set of tasks in such environments. Before actual systems of such vehicles, where no direct human control or other run-time intervention is required, can be successfully deployed, however, effective and scalable prototypes need to be designed, tested and evaluated. Agent-based modeling and simulation seem the most natural candidates for developing the needed mathematical and computational models in that context.

Various theories of *situated agents* (e.g., [12]) address the relationship, or coupling, between reactive, persistent agents embedded and acting in (generally, dynamic) environments, and their environments. Reactive situated agents need not be self-interested, and even if they are, they need not have any complex goals. However, even for such reactive situated agents that are relatively simple in that they need not be *deliberative*, it has been realized that (i) having a model of agent's environment and its dynamic behavior, and (ii) understanding the coupling and interaction between agent's actions or moves, and the corresponding moves (changes) in the environment, are of utmost importance [12]. In particular, theories of situated agents rightly treat the agent's environment as a "first class citizen".

In case of the more complex, deliberative agents, one of the causes of the additional complexity is the nature of an agent's goals, and the agent's need to deliberately affect the environment in order to be successful with respect to accomplishing its goals. The agent affects the environment by changing the environment's state via appropriately choosing among the available actions. Therefore we argue that, in case of such deliberative agents, both the environment and the agent's goals deserve "first class status".

An agent's goal-driven autonomous decision making needs to be expressible in terms of the most relevant parameters characterizing the agent's environment and goals. This autonomous decision making model specifies how is the agent to act, given its current internal state and the current state of the environment. In addition, the relationship among the agent's goals and its "measure(s) of success" on one, and the critical parameters of the environment, on the other hand, also needs to be established. That is, a complete model of such deliberative agents needs to capture the three-way coupling and interaction among (i) the agent's internal states and decision making mechanisms, (ii) the states and parameters of the "outside world", and (iii) the agent's goals, and some metrics of how successful the agent has been in accomplishing, or getting close to accomplishing, those goals.

This short paper has two main purposes. First, we outline a generic, broadly applicable "skeleton" model of agents' environments in terms of the basic parameters characterizing such environments. These parameters are also critical for each agent's goal-driven or utility-driven decision making process.

The parameters we identify are shared by most of the *MAS* applications we are aware of. We therefore hope that our discussion will provide some general guidelines to multi-agent system designers in different application areas, and assist them in identifying the most critical system parameters.

Secondly, we formulate a generic meta-problem of an autonomous agent acting in a multi-agent, multi-task dynamic environment, and point out some limitations of the classical decision theory when applied to an agent's action selection in scenarios where the agent's environment is sufficiently complex. The main contribution of this part of the paper is to identify some general types of scenarios where the classical decision theory likely should not be applied to modeling an agent's autonomous decision making. Some alternative, simple local-knowledge based (and therefore scalable) decision theoretic flavored mathematical models for an agent's action selection will be presented in our second paper in this volume.

2 Main Parameters of Agent Environments

An autonomous agent, whether natural or artificial, is often defined as an entity situated in and being a part of an environment ("world") such that it both can affect, and in turn be affected, by its environment [14]. A deliberative agent is an agent that is not merely reactive, but also has some notion of its purpose or goal(s) [14, 23]. The most common measure of an agent's successfulness with respect to that purpose or those goals is provided by an appropriate evaluation or utility function. Thus a notion of a deliberative autonomous agent that we adopt herein is that of an *autonomously acting entity* that is *reactive* and *persistent*, but also *pro-active* and *goal- or utility-driven* [19].

A pro-active and utility-driven agent accomplishes its goals and increases its utility by acting in its environment and, in particular, *changing* some properties of the environment. In order to effectively act upon the environment, the agent needs some mechanism that enables it, given the state of the environment and the agent's internal state, to effectively choose, among a set of available actions, a particular action that the agent hopes would affect the environment in a most desirable way in terms of its implications for the agent's agenda such as, e.g., this agent's (*individual*) *utility*. An agent's internal state typically captures some knowledge about the past states of the world, and how desirable different past states have turned out to be. In particular, an agent may have an internal representation of partial or complete histories of the previous states of the environment.

These considerations have two major implications. One, the agent correlates its measure of successfulness to the expected, or anticipated, future states of the environment that it thinks would result from undertaking particular actions. Two, the nature of an agent's environment or, more properly, agent's (quantitative, parametric) model of its environment, coupled with a model of how the agent's actions are expected to change the state of the environment, are critical to the agent's goal-driven action selection decision-making pro-

cess. In order to effectively design and/or analyze a deliberative, goal-driven agent that is an autonomous decision maker, it is, consequently, critical to also design a parametric model of this agent's "world", how the agent interacts with it, and how changes that occur in this world, in turn, affect the agent, by, e.g., getting it closer to or farther away from its goals. The changes that occur in an agent's "world", in general, include both those that are caused by the agent and those that are not.

What are, then, the most general properties, or parameters, of the agent environments that are found in the greatest variety of various autonomous agent systems? First, there are certain individual goals, or tasks, that each agent strives to accomplish. In general, different goals or tasks may differ in (i) how valuable they are to each agent, and (ii) how difficult (or easy) they are for an agent to accomplish. Moreover, completing any task usually does not come for free: the agent needs to spend some resources in order to get the job - that is, the task(s) - done. At the very least, completing any nontrivial task takes some amount of time. The agent may or may not have sufficient resources to complete any given task. Furthermore, additional resources, external to the agent, may or may not be available for the agent to acquire (perhaps at a certain cost) in order to be able to accomplish the task(s) it is after. Last but not least, the agent's environment may contain other agents.

We argue that, in most if not all interesting *MAS* scenarios and applications, an agent's environment is characterized by the following necessary "ingredients":

- some number (at least one, possibly more) of mutually independent tasks, where each of these tasks may be made of several (not necessarily independent) subtasks;
- a certain number (or amount) of various resources or capabilities that may be required for the completion of tasks; and
- other agents.

We identify the following critical quantitative parameters characterizing such an environment of our agent, and therefore largely defining its goals and how successful the agent is with respect to accomplishing those goals:

- the number of (independent) tasks that (currently) exist in the system;
- the (current) value of each task for the agent;
- the (current) vector (an appropriate tuple) of resource requirements of each task;
- the (current) number of other agents in the system.

Another critical system parameter, more reasonably attributed to the agent itself than to the "outer" world, is

- the vector (tuple) of amounts of each resource that the agent currently has at its disposal; that is, the tuple of agent's *capabilities* to service various tasks (see, e.g., [17]).

We describe in *Section 3* a fairly generic multi-agent, multi-task framework where an agent's autonomous decision making is a capability used for the purpose of maximizing agent's individual utility or payoff defined in terms of the agent's successiveness in completing those tasks. To illustrate the applicability of our generic model, we shall outline in the follow-up paper (also in this volume) how to apply our generic

framework to a concrete *MAS* application domain - the system of unmanned vehicles on a multi-task mission [5, 19].

We would like to focus specifically on agents that are capable of individual decision making in non-trivial dynamic environments that are, in general, *non-episodic* and only *partially accessible* to the agents [14]. The non-episodic nature of an agent's environment makes the agent's iterative reinforcement learning of the optimal behavior difficult or even impossible. Partial (in)accessibility, that is, the existence of certain aspects of the environment that are relevant to, yet not within the reach of, an agent would force this agent to have to reason and act "in darkness", that is, under uncertainty. Thus, the decisions made by agents in such a complex environment may in general impact both the agents themselves and their environment in ways that are not necessarily easy (or even possible) to reliably predict or fully observe - or, indeed, to even learn, over time, how to reliably predict in the future.

3 Generic Problem Formulation: Agents, Tasks, Resources

We now propose a high-level framework for a kind of multi-agent, multi-task problems where an agent's capability of autonomous decision-making plays a central role. We find this framework to be general enough to capture many situations where one has an ensemble of highly autonomous agents and a collection of independent tasks that these agents need to complete *dynamically* and in an *on-line* manner.

Let us consider a collection of N *agents* that need to serve a collection of up to M *tasks*. Each task T_j has a dynamically changing *value function* associated with it, that we denote $V_j(t)$. An agent, A_i , is driven by the desire to increase its own (expected) utility, U_i , by consuming as much of *value* of different tasks as possible. In particular, an agent is not simply embedded into its environment, where it may undertake different actions merely as a *reaction* to the observed changes in the environment. Instead, the agent pro-actively seeks to improve its own well-being.

In order to be able to meaningfully and effectively pursue the increase in (expected) utility or payoff, the agent must have some idea of what its goals or tasks are, what actions are on its disposal in order to pursue those goals or tasks, and some estimated *utility function* associated with completing each of these tasks. This is not to say, that the agent needs to *a priori* know all of its tasks or all of those tasks' values. However, some *a priori*, i.e., *built-in*, basic knowledge about the agent's goals, and awareness of the available capabilities and resources for accomplishing those goals, have to exist. This awareness can be expected, in general, to evolve with time, as the agent goes along in exploring its environment, and learning more about the tasks, the available and/or required resources, and the other agents.

When an agent discovers a particular task, it gets attracted by that task's value. If the agent happens to be simultaneously aware of two or more tasks at a given time step¹, the

¹Agent's awareness of tasks' existence, and some, not necessarily accurate,

agent needs to decide which of the tasks is currently *most attractive* to it. The agent needs to choose one of finitely many possible actions at its disposal. In general, such actions may include, for instance, sending messages to other agents in order to solicit their help in terms of sharing capabilities and/or resources in order to service the tasks - an example of *cooperative coordination*. Many other kinds of actions that include some form of *interaction* with other agents can be considered, in addition to those actions that do not (directly) include other agents, but, instead, only an agent's interaction with other, non-agent aspects of the environment. Whatever the set of possible actions in a given situation may be, the basic idea is that, the agent acting (or trying to act) rationally², it is the agent's desire that this action would maximize the agent's (expected) utility.

In our simple yet fairly generic multi-agent, multi-task problem formulation and related models of agents' autonomous action selection, the set of actions at an agent's disposal will be considerably simplified, and reduced to deciding which task an agent that is currently free chooses to tackle next (see *Section 4*). That is, at the starting point in modeling and analysis of an agent's autonomous decision making, interactions of an agent with other agents are abstracted away. This is not to say, that the impact other agents may have on an agent is altogether ignored. How exactly is this impact captured in our model will be discussed in the follow-up paper (also in this volume); for more, see [19], as well.

A few more words are due on our simple model of an agent's environment. In essence, from an agent's perspective, the environment is (*entirely*) made of *tasks*, *resources*, and *other agents*. The total amount of the available value of all tasks in the system is assumed to be bounded at all times. Consequently, the agents can be expected to sometimes end up competing for this limited source of utility increase that they are after. This general scenario leads to *competitive coordination*, where agents typically need to *negotiate* how are the tasks and/or resources to be divided among them (e.g., [13]). Likewise, resources available to each agent are also bounded, and an agent's resources may or may not suffice for servicing any particular task.

In our current framework that is considerably motivated by a particular application in mind (see our other paper in this volume, and also [5, 19]), the agents, assuming they all initially have the same amount of the same type of resources available, are all *equally capable*. In particular, no agent specialization beyond the restrictions imposed by the bounds on available resources is considered. The tasks appear identical to all agents - except possibly for their (true or estimated) values and resource demands. We further assume that the tasks are *mutually independent* of one another. In particular, whenever an agent has several available choices, which task(s) it is going to select to service, and in what order, is driven by the

idea of those tasks' values are results of either the agent having sensed those tasks, or because it got the (not necessarily reliable) information about the tasks from other agents.

²... although often under the circumstances of limited and incomplete observability and knowledge, and therefore *bounded rationality* [18].

agent's appropriate estimate of those tasks' values, resource consumption requirements, and the estimated competition for those tasks - that is, by the agent's estimate of the tasks' expected utilities to the agent *only*.

In our model, agents are assumed to be *self-interested* by default, that is, without an explicit or implicit *incentive*, they are not going to be inclined to cooperate. Each agent's (default) *sole goal* is to maximize its own payoff. However, various forms of coordination (including cooperation and collaboration) may nonetheless arise. An explicit incentive for cooperative coordination is, for example, an agent's realization that, with its own resources alone, it cannot accomplish a particular highly desirable task, and therefore, such agent may wish to use whatever communication channels it has on its disposal, to contact other agents and offer collaboration. This phenomenon is well-known in *multi-player game theory* [8], where the incentive for forming coalitions is still purely egoistic - including the realization by the rational agent (player) that the spoils, if any, will have to be split with the participating collaborators. Distributed resource-based coalition formation in distributed problem solving type of *MAS* has been studied, e.g., in [17]. We address distributed coalition formation for *MAS* made of *self-interested agents* in [20].

A generic example of a self-interested agent having an implicit incentive to cooperate with other agents is any kind of a conflict resolution scenario where the agents, unless they coordinate, can expect mutual destruction and, therefore, no hope of fulfilling their goals. Concrete such examples are readily available, among other domains, in the context of transportation systems and, in particular, in the domains of air traffic control and aerial vehicles (e.g., [7]).

4 Agents as Decision Makers: Some Challenges and Limitations

We now outline the bare essence of the (classical) decision theory in the context of an autonomous agent's decision making problem of an appropriate *choice of action*. Ideally, an agent would always choose an action leading to its optimal performance. Realistically, however, in most situations an agent can only strive for an expected and/or estimated and/or approximately optimal action selection. Moreover, sometimes "best-effort" or even merely "good enough effort" choices are sought, due to a number of limiting factors, from partial observability of the world to bounded computational and other resources.

How is this action selection problem an agent faces to be succinctly mathematically formalized? Following [10, 11], we first observe that an agent, A_i , by performing one of the actions at its disposal, say action a_k , affects the environment. Let the state of the environment (or that part of it that is accessible to the agent) be denoted by s . Let the agent's desirability of any particular state of the environment be quantified by that state's *utility value* to agent A_i that we shall denote $U_i(s)$. Typically, an agent cannot uniquely determine the next state of the world by its action, as there are other factors, possibly in-

cluding actions of other agents, that also affect the future state of the world. That is, the outcomes of the agent's actions need not necessarily always be deterministic.

Let $Pr(s_l|a_k)$ be the *conditional probability* that, if A_i does a_k , the subsequent state of the world is s_l . Then, according to *classical decision theory* (see, e.g., [10] and references therein), the *expected utility* of selecting action a_k for agent A_i is given by

$$E[U(a_k)] = \sum_{s_l \in States} \{Pr(s_l|a_k) \cdot U(s_l)\} \quad (1)$$

Therefore, an optimally acting agent, whenever facing an action selection problem as outlined above, will choose an action $a_* \in Actions$ according to

$$a_* = \arg\{\max_{a_k} \sum_{s_l} \{Pr(s_l|a_k) \cdot U(s_l)\}\} \quad (2)$$

However, this approach, based on the idea that an agent would strive to choose the particular action that maximizes its expected utility, rests on a number of nontrivial assumptions that need not even approximately hold in an actual application. Hence, the entire approach may, in practice, dramatically fail in case of a number of important *MAS* scenarios, including many that fit well into the general model we have outlined in *Sections 2-3*.

Usually an agent can only partially observe the world. Hence, assuming the agent can determine what "states" of the world any given action can lead to, in reality these "states", s_l , are actually only sub-states of the full states of the world. Thus each s_l uniquely determines an equivalence class of the possible full states of the world. Assuming there is a single-valued utility function defined on the set of all *complete states* of the world, utility values $U(s_l)$ defined over equivalence classes corresponding to partial states s_l are clearly not crisp but fuzzy (and, in general, multi-valued) - with degree of fuzziness that may be (i) hard or even impossible for the agent to estimate, and (ii) potentially so great as to render the computation in *Equation (2)* meaningless. Likewise, computing (or even estimating or approximating) conditional probabilities $Pr(s_l|a_k)$ may be infeasible or even impossible. For instance, in cases where an agent's *a priori* knowledge about the environment is very limited, so that even the set of *complete states* of the world is not known, computing these probabilities exactly is impossible, and any attempt at estimating them may yield arbitrarily poor estimates. This applies in those situations, among other, where an agent first needs to *discover* (yet unknown) tasks and/or resources in the environment, prior to ordering somehow their importance and being able to assign utility values to them³.

A typical approach to designing an agent that is going to be capable of arriving at the "right" choice of action a_* in *Equation (2)* is to endow the agent with some memory for storing the past history (i.e., sequences of the past states of the environment), and a capability to perform some sort of *iterative, online learning* from this stored knowledge. That

³More precisely, the utility values are assigned to those states or sets of states of the world that correspond to (that is, are expected to result from) the agent choosing particular task(s) to service.

is, an agent may begin with perhaps very poor estimates of the needed utilities and conditional probabilities, and, consequently, with very bad choices of actions estimated to be optimal at time step t . We denote herein an agent's estimate or guess of the optimal strategy a_* at time t by a_*^t . The central idea is that, as the agent proceeds along, presumably it would (or, at least, in principle *could*) learn how to improve its estimates of the required conditional probabilities and the utilities of various (sub)states of the world, thereby yielding improved choices of a_*^t , until, eventually, as $t \rightarrow \infty$, its behavior converges to optimal: $a_*^t \rightarrow a_*$.

There are many known iterative learning algorithms that hope to accomplish this kind of convergence to an optimal decision making behavior. Among many such reinforcement learning approaches, perhaps the most studied are those based on the idea of *Q-learning* (e.g., [15, 21]). Under certain assumptions, it has been shown that *Q-learning*, in the long-run (that is, in the limit), indeed provably converges to optimal agent behavior. However, thus achieved convergence is often very slow in practice, and therefore infeasible. Moreover, in case of inherently non-episodic environments, there are fundamental limitations on how much can be learned at all, and hence considerable additional resources may be spent (or wasted) by the agent without any guarantees of improved performance, let alone eventual convergence to the optimal behavior.

It is precisely this kind of autonomous agents' environments - dynamic, partially inaccessible, and non-episodic - and the corresponding decision making problems that we have in mind, and whose simplified generic version we have outlined in *Sections 2-3* and reference [19]. Hence, a genuinely different approach to an agent's autonomous action selection modeling seems to be required⁴.

5 Summary

The subject of this paper are *autonomous agents* that are operating in complex multi-agent, multi-task, bounded-resource dynamic environments. We emphasize the importance of an agent's models of its environment and its goals, and what are the implications of the characteristics of the environment, captured by an appropriate set of (in general, time-varying) parameters, for the agent's autonomous decision making process.

The problem of an agent's autonomous decision making viewed as an appropriate *action selection*, we argue, cannot be divorced from the nature or model of agent's environment and agent's goals. Our agents are pro-active, goal-oriented entities acting in dynamic, partially observable, unpredictable environments, and striving to maximize their expected utilities or payoffs under the circumstances of bounded rationality and bounded resources. In order to model an autonomous agent's decision making and study the challenges to be expected in designing such decision-making autonomous agents, we briefly discuss promises as well as limitations of the classical decision theory.

⁴A class of such models is discussed in the follow-up paper to this one.

This, first part of our work focuses on some parametric models of the agent environments, and on some limitations of the classical decision theory when those environments are sufficiently complex. The environment's complexity chiefly stems from two frequently encountered in practice, and assumed throughout herein, properties. The first property of a typical complex environment is, how (in)accessible, that is, (un)observable, it is to the agents. The other source of difficulty in designing effective deliberative agents is the environment's dynamic nature and, in particular, its being *non-episodic*; this property dramatically limits how much an agent can learn to predict the outcomes of its actions in the future based on the observed environment's behavior and the agent's corresponding payoffs in the past.

In our successor paper (also in this volume), we shall focus on some decision theoretic flavored models of an agent's action selection that circumvents some of the conceptual and feasibility problems with the classical approaches. In our modeling framework in the follow-up paper, an agent opts for reasoning and decision making that is *strictly local*, not burdened with resource-consuming high-level presentations of the world or the costly planning or learning heuristics. Hence, the agent's local knowledge based decision making becomes sufficiently simple and scalable (and, therefore, feasible) to apply even in the scenarios where the environments are highly dynamic and very complex, where the agents have to act online and in real-time, and where the underlying multi-agent systems are potentially of a very large scale.

Acknowledgment: Many thanks to the members of the Open Systems Laboratory's TASK group. This work was supported by the *DARPA IPTO TASK Program* under the contract *F30602-00-2-0586*.

References

- [1] N. M. Avouris, L. Gasser (eds.), "Distributed Artificial Intelligence: Theory and Praxis", Euro Courses Comp. & Info. Sci. vol. 5, Kluwer Academic Publ., 1992
- [2] D. P. Bertsekas, "Dynamic Programming: Deterministic and Stochastic Models", Prentice-Hall, 1987
- [3] D. H. Cansever, "Incentive Control Strategies For Decision Problems With Parametric Uncertainties", Ph.D. thesis, Univ. of Illinois Urbana-Champaign, 1985
- [4] S. Franklin, A. Graesser, "Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents", Proc. 3rd Int'l Workshop on Agent Theories, Architectures & Languages, Springer-Verlag, 1996
- [5] M. Jang, S. Reddy, P. Tolic, L. Chen, G. Agha, "An Actor-based Simulation for Studying UAV Coordination", Proc. 15th Euro. Symp. Simul. (ESS '03), Delft, The Netherlands, October 2003
- [6] H. Kopetz, "Scheduling", *Chapter 18* in "Distributed Systems" (ed. S. Mullender), ACM Press and Addison-Wesley, 1993
- [7] J. K. Kuchar, L. C. Yang, "A review of conflict detection and resolution modeling methods", IEEE Trans. Intelligent Transportation Systems, vol. 1, December 2000
- [8] J. von Neumann, O. Morgenstern, "Theory of Games and Economic Behavior", Princeton Univ. Press, 1944
- [9] G. Owen, "Game Theory" (2nd ed.), Academic Press, 1982
- [10] S. Parsons, M. Wooldridge, "Game Theory and Decision Theory in Multi-Agent Systems", Int'l J. AAMAS, vol. 5, Kluwer, 2000
- [11] S. Parsons, M. Wooldridge, "An introduction to game theory and decision theory", in "Game theory and decision theory in agent-based systems", S. Parsons, P. Gmytrasiewicz, and M. J. Wooldridge (eds.), Kluwer, 2002
- [12] S. J. Rosenschein, L. P. Kaelbling, "A Situated View of Presentation and Control", Artificial Intelligence vol. 73, 1995
- [13] J. Rosenschein, G. Zlotkin, "Rules of Encounter: Designing Conventions for Automated Negotiations among Computers", The MIT Press, Cambridge, Massachusetts, 1994
- [14] S. Russell, P. Norvig, "Artificial Intelligence: A Modern Approach", 2nd ed., Prentice Hall Series in AI, 2003
- [15] T. W. Sandholm, R. H. Crites, "On multi-agent Q-learning in a semi-competitive domain", Proc. IJCAI-95 Workshop on Adaptation & Learning in MAS, Montreal, Canada, 1995
- [16] O. Shehory, S. Kraus, "Coalition formation among autonomous agents: Strategies and complexity", Proc. MAA-MAW'93, Neuchatel, Switzerland, 1993
- [17] O. Shehory, S. Kraus, "Task allocation via coalition formation among autonomous agents", Proc. 14th IJCAI-95, Montreal, August 1995
- [18] H. A. Simon, "Models of Man", J. Wiley & Sons, New York, 1957
- [19] P. Tolic, M. Jang, S. Reddy, J. Chia, L. Chen, G. Agha, "Modeling a System of UAVs on a Mission", Proc. SCI '03 (invited session), Orlando, Florida, July 2003
- [20] P. Tolic, G. Agha, "Maximal Clique Based Distributed Group Formation Algorithm for Autonomous Agent Coalitions", Proc. Workshop on Coalitions & Teams, AAMAS '04, New York City, New York, July 2004
- [21] C. J. C. H. Watkins, "Learning from delayed rewards", Ph.D. thesis, Cambridge Univ., 1989
- [22] G. Weiss (ed.), "Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence", The MIT Press, Cambridge, Massachusetts, 1999
- [23] M. Wooldridge, N. Jennings, "Intelligent Agents: Theory and Practice", Knowledge Engin. Rev., 1995

SOME MODELS FOR AUTONOMOUS AGENTS' ACTION SELECTION IN DYNAMIC PARTIALLY OBSERVABLE ENVIRONMENTS

Predrag T. Tosic and Gul A. Agha

Open Systems Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign
1334 Siebel Center for Computer Science, 201 N. Goodwin, Urbana, IL 61801, USA
Email: {p-tosic, agha}@cs.uiuc.edu

ABSTRACT

We study the resource-bounded autonomous agents acting in complex, dynamic and partially observable multi-agent and multi-task environments, and, in particular, the agent action selection problem in such environments. Designing effective autonomous decision making agents is particularly challenging, due to a number of demands that such environments pose in terms of the agents' necessary capabilities. We make an early attempt in modeling and designing agents for large-scale multi-agent systems and complex environments, where individual agent's behaviors are sufficiently simple to be scalable and applicable in real-time settings, and where the agents' coordination and self-organization capabilities can still make agents (both individually and as ensembles) highly effective. This short paper focuses on finding *scalable* models of an agent's local knowledge based individual behavior. We propose herein several simple mathematical models for an agent's local knowledge-based action selection. We illustrate the general ideas about bounded-resource autonomous agents acting in complex dynamic environments with a concrete application example: a modeling framework for the scalable simulation of a collection of autonomous unmanned aerial vehicles (UAVs) on a multi-task mission.

KEYWORDS

multi-agent systems, autonomous agents, resource-bounded agents, bounded rationality, decision theory

1 Introduction

The subject of this work are some simple, scalable and feasible parametric models for individual agent behavior in complex dynamic environments that include multiple agents, resources and tasks. We shall view an autonomous agent as a pro-active, goal-driven and self-interested decision maker. Its autonomous decision making, coupled with other capabilities, such as adaptability and coordination with other agents, enables the agent to meaningfully strive to maximize its individual expected payoff. This action selection decision-making process critically depends on (i) the properties of an agent's environment (and the agent's model of its environment), and (ii) the characteristics of agent's goals that are defined as an appropriate function of the main parameters of the environment.

Even when much of the environment is inaccessible to the agent, the agent may still need to learn and reason (make inferences) about various environment's properties - including those that it cannot directly observe. That is, the agent needs to make inferences and decisions *under uncertainty*, and possibly also in the presence of various sources of *noise*. However, such high level cognitive abilities often place considerable computational and resource burdens on the agent. Hence, a resource-bounded agent may, instead, choose to "think and act (strictly) locally", fully aware that such approach may lead to suboptimal behavior. Moreover, we want each agent to act as autonomously as possible with respect to other agents - yet to still be able to coordinate (both competitively and cooperatively) with other agents, should the circumstances warrant such coordination.

An agent's goal-driven autonomous decision making needs to be mathematically expressed in terms of the most relevant parameters characterizing the agent's environment and goals. Such an autonomous decision making model should specify how is the agent to act, given its current internal state and the current state of the environment. In addition, the relationship between the agent's goals and "the measure(s) of success" with the critical parameters of the environment also needs to be established. That is, a complete model of such deliberative agents needs to capture the three-way coupling and interaction among (i) the agent's internal states and decision making mechanisms, (ii) the states and parameters of the "outside world", and (iii) the agent's goals, and some metrics of how successful the agent has been thus far in accomplishing those goals.

Given a kind of applications we primarily have in mind (ensembles of autonomous robots, unmanned vehicles [7, 26, 27]), instead of restricting deliberative autonomous agents exclusively to computer programs *alone* (as it is done, e.g., in [5]) we are more inclusive as to what kinds of entities and systems can meet the criteria for deliberative autonomous agents. In particular, we allow the entities that, in addition to computing capabilities based on an appropriate computer program, may also possess various *sensors*, *communication links* and *effectors* for interaction and information exchange with their environments.

In this paper, we shall formulate a generic problem of an autonomous agent acting in a multi-agent, multi-task dynamic environment, and make an attempt to modify the classical decision theoretic action selection formulations in

order to obtain applicable, feasible and *scalable* parametric models for an agent's utility-driven action selection in such environments. The main idea is to trade the likely unreachable optimality for efficiency, practicality and locality as the primary metrics of how good and applicable in online and real-time settings (see [27]) an agent's action selection strategy is. We then briefly indicate how this general framework can be fruitfully applied to a concrete *MAS* application: a system of autonomous, resource-bounded *unmanned aerial vehicles (UAVs)* on a multi-task mission [7, 8, 26, 27].

2 Autonomous Agents' Individual Behavior Functions

We view *agent autonomy* as a *capability of pro-active, goal-driven decision making*. One of the central challenges in *MAS* research is how to model this decision-making process, and what are the critical parameters that it depends on? We have argued in a companion paper (also in this volume) that the common prescriptions of *classical decision theory* do not work well in situations where the agents' environments are sufficiently complex - in particular, when these environments are dynamic, non-episodic and (partially) inaccessible to agents. The general prescription for an agent's action selection, given by *Eqn. (2)* in the companion paper is problematic, due to intrinsic intractability or even impossibility of computing the needed conditional probabilities and utilities. An alternative approach, based on Markov games (e.g., [11]) and *Partially Observable Markov Decision Processes (POMDPs)*, (e.g., [6]) has been extensively studied by the *MAS* research community (see, e.g., [12, 13, 14]), but it is known that *POMDPs* are prohibitively computationally expensive except for the simplest cases and small-sized problems [2, 12].

We now try to justify the class of generic models of agent autonomy in the context of the general problem of an autonomous agent acting in an unknown or partially known, multi-agent, multi-task dynamic environment as outlined in the predecessor paper (also in this volume). The model of an agent's autonomous decision making will be an appropriate class of action-selecting mathematical functions, that, depending on the situation, in general can be deterministic, nondeterministic or probabilistic¹.

Herein, for simplicity we shall restrict the class of models of an individual agent's autonomous decision-making to action-selection mechanisms that are required to be *deterministic*, and that we shall call *individual behavior functions*, $\Theta_i(t)$ [26]. Given a set of tasks with their current values, an agent A_i evaluates its behavior function Θ_i that returns the index j^* of the task T_{j^*} such that, if A_i selects T_{j^*} as its next task to service, this choice,

¹That is, a *probability distribution* over a well-defined, *finite* set of possible actions, or, more generally, for autonomous agents more complex than what we consider herein, over a set of agent's possible plans or strategies.

from A_i 's perspective, ought to maximize the *estimated (expected) increase* in an appropriate utility-like quantity, U_i . A generalization of this short-term, "single-shot" action selection mechanism given by Θ_i to the individual behavior functions that, instead of a single index j^* , would return partial or complete (*expected*) *optimal or approximately optimal schedules* (j_1^*, \dots, j_k^*) , is immediate.

The parameters that these individual behavior functions in general may depend on include the estimated resources and/or capabilities that the agent A_i needs to service a particular task T_j , the task's current value $V_j(t)$ (or an imperfect estimate of this value), the task's resource requirements, and the estimated competition for that task and its value, i.e., the number of other agents that are estimated to be interested (or, possibly, needed to provide assistance to agent A_i) in servicing the task T_j .

Let $R_i^j(t)$ be the *resource demand vector* that represents how much of each of several pre-specified resources is agent A_i going to need in order to fully service task T_j (and thus, assuming no competition, to consume all of the still available task's value, V_j). Let $C_i(t)$ be the vector of *available resources (capabilities)* - these are the resources (capabilities) that agent the A_i "owns" and may freely decide whether and how to utilize (i.e., spend) in its pursuit of finding and servicing tasks and, consequently, thus increasing its individual utility, U_i . Let $\|R_i^j\|$ be the *norm* (or, more generally, some *cumulative function* of all components) of the resource demand vector R_i^j , and let $n_{j,r}$ be the (estimated) number of agents interested in task T_j according to some parametric criterion r . One general class of models of the i -th agent's target task selection can be specified by

$$\Theta_i(t) = \arg \{ \max_{1 \leq j \leq M} G(V_j, R_i^j, C_i, n_{j,r}, t) \} \quad (1)$$

where G is a function that is increasing in the current (true or estimated) task's value, V_j , and non-increasing in the resource requirements R_i^j .

One example of a simple individual behavior that fits the general framework specified by *Eqn. (1)* is given by

$$\Theta_i(t) = \arg \{ \max_{1 \leq j \leq M} \left[\frac{V_j(t)}{(n_{j,r}(t) + 1) \cdot \|R_i^j\|} \right] \} \quad (2)$$

where it is assumed that the norm of resource requirements of any task with respect to any agent is strictly positive.

Let us assume that, in the given concrete multi-agent, multi-task problem, *time* to complete different tasks is the most critical resource. Then one may want to use the agent choice-of-action function Θ_i given by

$$\Theta_i(t) = \arg \{ \max_{1 \leq j \leq M} \left[\frac{V_j(t) - t_{est}^j \cdot n_{j,r}(t) \cdot d_j}{n_{j,r}(t) + 1} \right] \} \quad (3)$$

where t_{est}^j is the *estimated time* that agent A_i would need to spend in pre-processing, before starting consuming a particular task's value V_j , and where d_j stands for the *value consumption rate* for T_j . The tacit assumptions

made in *Eqn. (3)* - such as, for example, that each agent, from the moment it starts consuming the task's value, will get an "equal share" of the remaining value as any other agent working on the same task - are oversimplifications that may not hold in a given situation even as crude approximations. Yet, such simplifying assumptions may be often necessary, in order for the agent's choice-of-action function Θ_i to be readily (and *quickly*) computable "on the fly".

Should the agent's computing resources and the available time for deliberation allow it, the agent can readily incorporate online learning into the action selection mechanism given by *Eqn. (3)*. For instance, typically not all other agents that seem interested in task T_j will actually pursue this particular task. Hence, a discount parameter $\gamma(t)$ such that $0 \leq \gamma(t) \leq 1$, that indicates what fraction of other agents that seem interested in task T_j , is likely going to participate in servicing this task, can perhaps be learned over time, and the action selection in *Eqn. (3)* appropriately modified:

$$\Theta_i(t) = \arg\{\max\left[\frac{V_j(t) - t_{est}^j \cdot \gamma(t) \cdot n_{j,r}(t) \cdot d_j}{\gamma(t) \cdot n_{j,r}(t) + 1}\right]\} \quad (4)$$

Similarly, optimal or good values of other parameters appearing in various action selection models, in principle, can be also learned - assuming the sufficient time and other resources at the agent's disposal.

We begin the discussion of the proposed agent action-selection problem with a remark on some work seemingly similar to ours. The model of the environment in [24], and, in particular, the model of tasks with their resource requirements, are the same as ours. The problem addressed, namely, distributed task allocation, also appears quite similar to our agent action-selection problem. However, the agents in [24] are not self-interested, have no individual agendas or utilities, and are jointly attempting to maximize the *joint* (or *common*) *utility*. Thus [24] is about *distributed problem solving*, whereas the fundamental characteristic of our agents is that they are self-interested.

Let us now initially assume that the agents have *perfectly reliable* (within their finite ranges) sensors and communication links. An agent's knowledge of the environment, and of the tasks in particular, even if assumed (locally) accurate, is still, in general, not complete. Due to the ontological assumptions of (i) no central control, and (ii) bounded sensor and communication ranges, each agent necessarily has only a *local* picture of the tasks, as well as of the other agents and their whereabouts. If the agents work in unison, i.e., if they have the common goal that they are striving to achieve (that is, a single *joint utility function*), then the problem of how to split up the tasks among the agents, and in what order, approximately reduces to a well-known problem of (*distributed*) *online task allocation and scheduling* (e.g., [24, 9]).

However, the problem we desire to model is considerably more complex than mere distributed dynamic schedul-

ing. The main reason behind this additional complexity is the inherently distributed and local nature of individual agents' information, knowledge, and interests. Even under the (usually unrealistic) assumptions of perfectly reliable communication and sensing, we identify the following generic sources of additional difficulty: (i) each agent only has a local knowledge of the tasks and of other agents (and of the world in general); in particular, an agent can be affected by those aspects or "parts" of the world that it itself cannot access or directly influence; (ii) each agent is trying to optimize its own individual payoff, and there is no guarantee, in case of those application domains where the overall system performance actually matters, that individual self-interested efficiency would necessarily lead to a satisfactory efficiency of the system as a whole; (iii) even if communication links are perfectly reliable, the information that an agent receives from other agents need not be reliable, as the agents, in general, can be expected to compete for tasks and therefore the *veracity* assumption, in any such competitive scenario, need not hold.

Observation (i), and models for and analysis of this paradigm in various application domains, as well as the design of agents that can overcome the adversities due to this limitation, are the central subject of *distributed artificial intelligence* [31]. Similarly, (ii) and its generalization - how to reconcile the quest for maximizing individual vs. joint utility functions - is a subject matter of either *incentive engineering* [4], in case that an agent itself has to ensure appropriately defined system efficiency (for instance, *social welfare*) while pursuing its own agenda and self-interest, or of *mechanism design* [21], in case where an outside intelligent entity (typically, the designer of the *MAS*) ensures that self-interested agents would "play by the rules" as this actually is in their individual best interest, and they are sufficiently rational to realize that. Finally, (iii) takes us into the realm of (*many-player*) *game theory* (e.g., [19, 15]).

Thus far we have assumed that the agents' knowledge of their environment and their tasks, while local, is perfectly accurate, in that the sub-state of the (complete) state of the world that is accessible to the agent², s_l , is reliably and accurately known by the agent. Most of the time, a realistic agent model has to drop the assumption of reliable local knowledge in favor of an imperfect, noisy model of an agent's knowledge of even those aspects of the world that are fully accessible to the agent. In general, an agent's sensors cannot be assumed to be perfectly accurate: whatever properties of the environment they measure, these measurements likely introduce uncertainty and noise in the agent's local picture of the world. Likewise, an agent's communication links can be seldom assumed perfectly reliable: they may be faulty and they may experience delays. The delays in communication may cause an agent to base its decisions on outdated information about the world, which may lead to potentially catastrophic consequences.

²See detailed discussion in the companion paper immediately preceding this paper.

3 An Application

A collection of *Unmanned Aerial Vehicles (UAVs)* on a multi-task mission provides a suitable framework for identifying, modeling and analyzing many interesting paradigms, design parameters and solution strategies applicable not only specifically to autonomous unmanned vehicles and teams of robots, but to *Multi-Agent Systems (MAS)* in general. UAVs are already finding, or are anticipated to find, their use in a variety of military and law-enforcement operations, e.g., in various surveillance, reconnaissance, and search-and-rescue tasks. A typical UAV or micro-UAV is equipped with certain *sensors* (such as, e.g., radars or infra-red cameras). With these sensors, a UAV probes its environment and forms a (local) “picture of the world” on which its future actions may need to be based. A UAV is also equipped with some *communication capabilities*, that enable it to communicate with other UAVs and/or the ground or satellite control. This communication enables a UAV to have an access to the information that is not local to it - that is, the information not directly accessible to the UAV’s sensors.

While trying to accomplish their mission, these UAVs need to respect a heterogeneous set of constraints on their physical and communication resources. The UAVs also need to be able to communicate and coordinate with each other. Their cooperative coordination may range from merely assuring that they stay out of each other’s way (collision avoidance) to enabling themselves to adaptively and dynamically divide-and-conquer their tasks.³

Not all kinds of UAVs can be reasonably considered genuine autonomous agents; e.g., those that are remotely controlled throughout their mission are neither autonomous nor agent-like. However, for the reasons of system scalability, dependability and robustness, increasingly complex and autonomous unmanned vehicles are being studied and designed. We are interested in (micro-)UAVs that are not remotely controlled and that have the ability to make their own decisions in real time. We are also assuming, for the most part, no central control of any sort (see [7, 26, 27]). In particular, the knowledge of the world that each UAV possesses is assumed to be *local*, possibly *noisy*, to vary with time, and to be augmentable, at a certain cost, via communication with other UAVs.

Some of the problems that have been extensively studied in the context of UAVs include motion planning and conflict detection and resolution; see, e.g., [3, 10, 16]. What has drawn considerably less attention (until very recently) is modeling and analysis of the goal-driven or utility-driven autonomous behavior of the UAVs that can be reasonably viewed as autonomous agents [7, 26, 27].

We now turn to a *MAS* formulation of a system of autonomous UAVs and how is each UAV to choose tasks or targets. A collection of N UAVs needs to accomplish

a certain complex, multi-task mission. We model this mission as a set of M tasks, or *interest points (IPs)* [26]. Each interest point T_j has a dynamically changing value associated with it, $\Pi_j(t)$. An IP may be static or mobile. A mobile IP T_j , at any time step t , is completely and uniquely specified by its position and velocity vectors, $\psi_j(t)$ and $\xi_j(t)$, respectively, and its value $\Pi_j(t)$ [7, 26]. This model can be readily augmented by adding a time-dependent resource requirement vector to each IP, in lieu with the generic model in *Section 2*. Each UAV V_i is driven by the desire to increase its own utility, U_i , by consuming as much of value of various IPs as possible. The total amount of value is assumed to be bounded at all times. Consequently, the UAVs can be expected to compete for this limited resource.

From an individual UAV’s perspective, the goal is to maximize its own utility, by visiting as many interest points and consuming as much of their value as possible. This is accomplished by following a certain either fixed or dynamically changing (adaptable) *individual behavior strategy*. This individual behavior can be specified by an appropriate *individual behavior function*, Θ_i , that UAV V_i follows as long as there is no outside signal telling the UAV it should start doing something else. An example of such outside signal is a request to a given UAV to join a newly formed group. If such a request comes from a leader whose supremacy in authority is recognized, the follower UAV will have to abandon its current behavior and comply with the leader’s desires, thereby giving up its individual autonomy. Thus, one can observe an instance of a fundamental *tradeoff* between individual autonomy and group coordination which may require (partial or complete, temporary or permanent) sacrifice of the agent’s autonomy.

For much more on modeling UAVs as autonomous agents, on the nature of interest points, and some models of multi-agent coordination (as well as the interaction between individual autonomy and multi-agent coordination), we refer the reader to [26, 27]. A UAV simulation testbed is described in some detail in [7, 8].

4 Some Simple Models for UAV Autonomous Action Selection

In order for any type of *an unmanned vehicles* to be considered *autonomous agents*, they have to be capable of *autonomous decision making under uncertainty* without direct intervention by an outside operator. We outline a simple model of autonomy applicable to UAVs that would render UAVs proper autonomous agents. Herein, UAVs are modeled as *utility-driven entities*. They fulfill their goals and thus increase their utilities by servicing their tasks; we refer to these UAVs’ tasks as *interest points (IPs)*, and associate an appropriate, time-dependent *value function* Π_j to each IP. As we assume that a single UAV can consume value from at most one IP at a time, the question arises: among several candidate IPs, how should a UAV choose in

³In [26], this latter, higher form of cooperative coordination we also call *goal-driven* (cooperative) coordination.

what order it is to visit these IPs? Therefore, each UAV faces an *online scheduling problem*. We further simplify the analysis, and only ask, given a set of interest points whose current positions and (estimated) values are known to a particular UAV, which IP among them should the UAV select to visit next?

Given a set of IPs with their current positions and values⁴, a UAV V_i evaluates its behavior function Θ_i that returns the index j^* of the IP such that, if the UAV selects that IP as its next task to service, this choice, from that UAV's perspective, is expected to maximize the *estimated increase* in the UAV's *utility*. In particular, each UAV is assumed *self-interested* - unless and until ordered differently.

Clearly, a great variety of self-interested individual strategies can be specified via different choices of the functions Θ_i .

We can now readily specialize the general model from the previous sections to the concrete application domain at hand. Some parameters that individual behavior functions of UAVs can be expected to depend on are the UAV's distance from the given IP, the IP's current value (or its estimate), the UAV's currently available resources (or capabilities), the IP's resource requirements, and the estimated competition for that IP and its value - viz., the number of other UAVs in the IP's vicinity. Let $x_i(t)$ and $v_i(t)$ be the i -th UAV's position and velocity vectors at time t , respectively, let $\psi_j(t)$ be the position of IP j at time t , $\xi_j(t)$ its velocity, and let $n_{j,r}$ be the total number of UAVs within the distance r from IP j . Then one class of models of the i -th UAV's target selection strategy can be specified by

$$\Theta_i(t) = \arg\{\max_j G(\Pi_j, \|x_i - \psi_j\|, \|v_i - \xi_j\|, n_{j,r}, t)\} \quad (5)$$

where G is a function that is increasing in Π_j and non-increasing in the distance of the UAV from the IP j , $\|x_i(t) - \psi_j(t)\|$, and the (estimated) relative velocity between the two, $\|v_i(t) - \xi_j(t)\|$.

One example of a simple individual behavior that fits the given general framework and that we have considered in case of *fixed (static) IPs* [26] is given by

$$\Theta_i(t) = \arg\{\max_{1 \leq j \leq M} [\frac{\Pi_j(t)}{n_{j,r}(t) \cdot \|x_i(t) - \psi_j\|}]\} \quad (6)$$

where there exists a strictly positive minimal allowable distance of any UAV from any IP, and where distances are appropriately normalized so that one may treat them as (physically) "dimensionless".

Another example of a simple yet not entirely trivial greedy individual behavior, that assumes the IPs are either stationary or else that their velocities can be neglected, is given by

$$\Theta_i(t) = \arg\{\max_{1 \leq j \leq M} [\frac{\Pi_j(t) - n_{j,r}(t) \cdot d}{\|x_i(t) - \psi_j\|}]\} \quad (7)$$

⁴In case of *mobile targets (IPs)*, current velocities of the targets (IPs) would be also needed.

where d above stands for the (*constant*) *consumption rate* of an IP's value, and where similar assumptions hold as before.

Yet another, slightly more elaborate UAV choice-of-action function that we have considered is given by

$$\Theta_i(t) = \arg\{\max_{1 \leq j \leq M} [\frac{\Pi_j(t) - t_{est}^j \cdot n_{j,r} \cdot d}{n_{j,r} + 1}]\} \quad (8)$$

where t_{est}^j is the *estimated time* that UAV V_i will need to reach close enough to IP T_j in order to start consuming its value. Thus *Eqn. (8)* above can be considered as a special case of *Eqn. (3)* in *Section 2*.

Everything said thus far about UAVs' individual behavior strategies rests on the assumption that each UAV acts strictly selfishly, and largely independently (except for the dependence of Θ_i on $n_{j,r}$) from what other UAVs do. Once UAV-to-UAV communication and coordination are taken into account, faithfully modeling a UAV's autonomous behavior becomes considerably more complex. In particular, in addition to the already mentioned parameters, each UAV's "picture of the world" and, consequently, its $\Theta_i(t)$ would be expected to also depend on messages that V_i has received from other UAVs.

An important point about the IP value function $\Pi_j(t)$ is that this function, for each IP T_j , represents the T_j 's true (or objective) value, irrespective of which UAV may have observed T_j , and from how far away [7, 26]. The tacit assumption has been that, once an IP is discovered by the UAVs, all UAVs who are aware of this IP's existence immediately also know its exact (current) value. A more realistic assumption is that each UAV has its own, *local* and, in general, *imperfect* (i.e., *noisy*) and intrinsically *uncertain* knowledge of each of IPs' values.

5 Summary and Conclusions

The subject of this paper are *autonomous agents* that are operating in complex multi-agent, multi-task, bounded-resource dynamic environments. We emphasize the importance of what are an agent's models of its environment and its goals, and how the properties of the environment crucially affect an agent's autonomous decision making process.

The problem of an agent's autonomous action selection cannot be divorced from the nature and model of the agent's environment and the agent's goals. Our agents are pro-active, goal-oriented entities acting in dynamic, partially observable, unpredictable environments, and striving to maximize their expected utilities under the circumstances of bounded rationality and bounded resources. In that context, we consider promises as well as limitations of the classical decision theory. The first part of our work focuses on some parametric models of agent environments, and on limitations of classical decision theory when those environments are sufficiently complex (cf. in terms of

how (in)accessible, that is, (un)observable, they are to the agents).

We propose herewith a class of generic models for an agent's task selection applicable to multi-agent, multi-task complex dynamic environments, where the choice of action can be abstracted to selecting (or scheduling) one or more among the available tasks to be serviced. The main idea is to trade complexity and poor scalability (that tend to come hand-in-hand with a quest for optimality) for simplicity, high scalability and locality in this decision making process. To illustrate the usefulness of our simple models for the design of application-level agents that are online decision makers in actual dynamic, non-episodic and partially inaccessible environments, we indicate in *Sections 3-4* how the generic framework and task-selection models can be applied to a system of unmanned autonomous vehicles on a multi-task mission.

As for the future work, we envision two main directions. One is to consider more complex individual behavior strategies of agents. In particular, introducing agent's memories, and subsequently the ability of an agent to learn from and reason about its past experiences, could make agents more effective in terms of making good decisions, while less prone to instability due to "knee-jerk" reactive responses to frequent and unpredictable random oscillations in the environment. However, such reasoning would place considerable additional cognitive and resource consumption burdens on the agent, thereby possibly hurting the model's scalability and hence applicability to very large scale *MAS*. Finding an appropriate tradeoff between the complexity and sophistication of an agent's deliberation model, and the system's scalability and efficiency, is a task of utmost importance. Clearly, no "one size fits all" single solution is possible, and a *MAS* system designer needs to make her own choices based on the properties of the application at hand, available resources, and other application-dependent considerations.

The second important consideration that needs to be further addressed is the interaction, synergy and conflict between an agent's self-interest-driven individual decision making, and multi-agent (cooperative and/or non-cooperative) coordination.

While the models of autonomous agents, their environments and their decision making presented herein do require a considerable future work on both further generalization as well as application to concrete problem domains, we do hope that this report sheds some light and makes a modest contribution to understanding and modeling autonomous agents acting in complex - that is, multi-agent, multi-task, bounded-resource, partially inaccessible, unpredictable and non-episodic - and dynamic environments.

Acknowledgment: This work was supported by the DARPA IPTO TASK Program contract # F30602-00-2-0586.

References

- [1] N. M. Avouris, L. Gasser (eds.), "Distributed Artificial Intelligence: Theory and Praxis", Euro Courses Comp. & Info. Sci. vol. 5, Kluwer Academic Publ., 1992
- [2] D. P. Bertsekas, "Dynamic Programming: Deterministic and Stochastic Models", Prentice-Hall, 1987
- [3] A. Bicchi, L. Pallottino, "On optimal cooperative conflict resolution for air traffic management systems", IEEE Trans. Intelligent Transport. Systems, vol. 1, Dec. 2000
- [4] D. H. Cansever, "Incentive Control Strategies For Decision Problems With Parametric Uncertainties", Ph.D. thesis, Univ. of Illinois at Urbana-Champaign, 1985
- [5] S. Franklin, A. Graesser, "Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents", Proc. 3rd Int'l Workshop on Agent Theories, Architectures & Languages, Springer-Verlag, 1996
- [6] R. A. Howard, "Dynamic Programming and Markov Processes", The MIT Press, Cambridge, MA, 1960
- [7] M. Jang, S. Reddy, P. Totic, L. Chen, G. Agha, "An Actor-based Simulation for Studying UAV Coordination", Proc. 15th Euro. Symp. Simul. (ESS '03), Delft, The Netherlands, October 2003
- [8] M. Jang, G. Agha, "On Efficient Communication and Service Agent Discovery in Multi-agent Systems," 3rd Int'l Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS '04), pp. 27-33, May 24-25, Edinburgh, Scotland, 2004
- [9] H. Kopetz, "Scheduling", Chapter 18 in "Distributed Systems" (ed. S. Mullender), ACM Press and Addison-Wesley, 1993
- [10] J. K. Kuchar, L. C. Yang, "A review of conflict detection and resolution modeling methods", IEEE Trans. Intelligent Transportation Systems, vol. 1, December 2000
- [11] M. Littman, "Markov Games as a Framework for Multi-Agent Reinforcement Learning", Proc. 11th Int'l Conf. Machine Learning (ML'94), 1994
- [12] M. L. Littman, A. Cassandra, L. P. Kaelbling, "Learning policies for partially observable environments: Scaling up", Proc. 12th Int'l Conf. Machine Learning, San Francisco, CA, 1995
- [13] W. S. Lovejoy, "A survey of algorithmic methods for partially observed Markov decision processes", Annals of Oper. Research vol. 28, April 1991
- [14] G. E. Monahan, "A survey of partially observable Markov decision processes: Theory, models, and algorithms", Management Sci., vol 28 (1), 1982
- [15] G. Owen, "Game Theory" (2nd ed.), Academic Press, 1982
- [16] L. Pallottino, E. M. Feron, A. Bicchi, "Conflict Resolution Problems for Air Traffic Management Systems Solved With Mixed Integer Programming", IEEE Trans. Intelligent Transportation Systems, vol. 3, No. 1, March 2002
- [17] S. Parsons, M. Wooldridge, "Game Theory and Decision Theory in Multi-Agent Systems", Int'l J. AAMAS, vol. 5, Kluwer, 2000
- [18] S. Parsons, M. Wooldridge, "An introduction to game theory and decision theory", in "Game theory and decision theory in agent-based systems", S. Parsons, P. Gmytrasiewicz, and M. J. Wooldridge (eds.), Kluwer, 2002
- [19] A. Rapoport, "N-Person Game Theory", The Univ. of Michigan Press, 1970
- [20] S. J. Rosenschein, L. P. Kaelbling, "A Situated View of Presentation and Control", Artificial Intelligence vol. 73, 1995
- [21] J. Rosenschein, G. Zlotkin, "Rules of Encounter: Designing Conventions for Automated Negotiations among Computers", The MIT Press, Cambridge, Massachusetts, 1994
- [22] S. Russell, P. Norvig, "Artificial Intelligence: A Modern Approach", 2nd ed., Prentice Hall Series in AI, 2003
- [23] T. W. Sandholm, R. H. Crites, "On multi-agent Q-learning in a semi-competitive domain", Proc. IJCAI-95 Workshop on Adaptation & Learning in MAS, Montreal, Canada, 1995
- [24] O. Shehory, S. Kraus, "Task allocation via coalition formation among autonomous agents", Proc. 14th IJCAI-95, Montreal, August 1995
- [25] H. A. Simon, "Models of Man", J. Wiley & Sons, New York, 1957
- [26] P. Totic, M. Jang, S. Reddy, J. Chia, L. Chen, G. Agha, "Modeling a System of UAVs on a Mission", Proc. SCI '03 (invited session), Orlando, Florida, July 27-30, 2003
- [27] P. Totic, G. Agha, "Modeling Agent's Autonomous Decision Making in Multi-Agent, Multi-Task Environments", Proc. 1st Euro. Workshop on Multi-Agent Systems (EUMAS'03), Oxford, England, December 18-19, 2003
- [28] P. Totic, G. Agha, "Maximal Clique Based Distributed Group Formation Algorithm for Autonomous Agent Coalitions", Proc. Workshop on Coalitions & Teams, AAMAS '04, New York City, New York, July 19-23, 2004
- [29] C. J. C. H. Watkins, "Learning from delayed rewards", Ph.D. thesis, Cambridge Univ., 1989
- [30] C. J. C. H. Watkins, P. Dayan, "Q-Learning", Machine Learning vol. 8 (3), 1992
- [31] G. Weiss (ed.), "Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence", The MIT Press, Cambridge, Massachusetts, 1999
- [32] M. Wooldridge, N. Jennings, "Intelligent Agents: Theory and Practice", Knowledge Engin. Rev., 1995

Task Assignment for a Physical Agent Team via a Dynamic Forward/Reverse Auction Mechanism

Amr Ahmed, Abhilash Patel, Tom Brown, MyungJoo Ham, Myeong-Wuk Jang, Gul Agha

Open Systems Laboratory

Department of Computer Science

Urbana, IL 61802, USA

{amrmomen,apatel1,tdbrown,ham1,mjang,agh}@uiuc.edu

<http://osl.cs.uiuc.edu>

Abstract — In the dynamic distributed task assignment (DDTA) problem, a team of agents is required to accomplish a set of tasks while maximizing the overall team utility. An effective solution to this problem needs to address two closely related questions: first, how to find a near-optimal assignment from agents to tasks under resource constraints, and second, how to efficiently maintain the optimality of the assignment over time. We address the first problem by extending an existing **forward/reverse auction** algorithm which was designed for bipartite maximal matching to find an initial near-optimal assignment. An important problem with such assignments is that the **dynamicity** of the environment compromises the optimality of the initial solution. We address the dynamicity problem by using **swapping** to locally move agents between tasks. By linking these local **swaps**, the current assignment is morphed into one which is closer to what would have been obtained if we had re-executed the computationally more expensive auction algorithm. In this paper, we detail the application of this dynamic auctioning scheme in the context of a UAV (Unmanned Aerial Vehicle) search and rescue mission and present early experimentations using physical agents to show the feasibility of the proposed approach.

1. INTRODUCTION

Recently, the problem of dynamic distributed task assignment (DDTA) among a team of agents has gained tremendous attention due to the wide variety of applications that require an efficient solution to this problem like: distributed sensor network [2], vehicle monitoring [4], and search and rescue [6]. In the DDTA problem a team of agents is required to accomplish a set of tasks according to a given criteria. This criterion can be either minimizing the time to accomplish all tasks or maximizing the utility of the accomplished tasks in a given time frame.

Several approaches have been proposed to solve this

problem that can be classified as either centralized or distributed. In centralized approaches [10], there exists a central agent who plays the role of a leader. This leader aggregates information from other team members, plans optimally for the entire team, and finally propagates the task assignments to other team members. This master-slave architecture has the advantage of finding an optimal solution, yet it has several disadvantages such as: a single point of failure, inability to respond fast to changes in the environments, and inability to deal with partially observable environments.

To deal with the above shortcomings, distributed approaches were proposed. They attack the DDTA problem by requiring each agent to plan for itself based on local information [11]. In these approaches agents rely on a predefined negotiation framework that allows them to decide what activity to do next, what information to communicate, and to whom. A difficulty with these approaches is that they require agents to possess accurate knowledge about their environment which is difficult to maintain in heterogeneous and open systems.

What is missing from the previous approaches, as reported in [2], is a formalization of the DDTA problem that exposes its challenging requirements and drives researchers to design efficient algorithms for this important problem. These efficient algorithms need to address two closely related questions: 1) *combinatorial issues*: how to find a near-optimal assignment from agents to tasks under time and bandwidth resource constraints, and 2) *environment dynamicity*: how to efficiently maintain the optimality of the assignment over time.

In this paper we present a dynamic auctioning scheme that uses a divide and conquer strategy to approach the DDTA problem. We address the first question by extending a forward/reverse auction algorithm [1] which was originally designed for bipartite maximal matching to handle non-unary task requirements. This algorithm alternates between rounds of forward and reverse auctions. In the forward stage, agents bid for tasks, while in the reverse stage tasks conceptually bid for agents by reducing their prices. Because the environment is dynamic the solution found during the auction may degrade from optimal to highly inefficient, we propose to use *swapping* to locally move agents between tasks. By linking these local *swaps*,

the current assignment is morphed into the one which should have been obtained if we had re-executed the expensive auction algorithm.

The rest of this paper is organized as follows. The next section describes our application domain, i.e., the UAV (Unmanned Aerial Vehicle) search and rescue domain. In section 3 and 4 we detail the use of the dynamic auctioning scheme within this domain. Section 5 describes the UAV agent architecture, while section 6 presents our flexible experimental setting and reports on early experiments with this domain. In section 7 we discuss related work, and finally in sections 8 we conclude this paper and list several future research directions.

2. THE APPLICATION DOMAIN

We used a search and rescue mission as an example of the DDTA problem. In this application domain, a set of UAVs¹ roam a rectangle mission area looking for targets (downed pilots, injured civilians, etc). These targets move according to a pre-determined path not known to the UAVs. Each target has a step utility function as depicted in Figure 1 and requires a minimum number of UAVs to be serviced. This step utility function means that before the target gets its required number of UAVs, none of its utility can be consumed by the team. Once the required number of UAVs arrived around the target, it will be considered to be serviced. UAVs monitor targets and coordinate to form groups to service them subject to maximizing the total team benefit as described by Equation 1:

$$\max \sum_{t \in \text{targets}} \left(\text{util}_t - \sum_{a \in \text{group}(t)} c_{at} \right) \dots (1)$$

where $\text{group}(t)$ is the set of UAVs assigned to target t , util_t is the utility value of target t , and c_{at} is the cost incurred by UAV a to service target t (in our scenario, this is the cost of the path length along which the UAV moves to the target).

3. THE FORWARD / REVERSE AUCTION

The forward/reverse auction algorithm was originally proposed by Bertsekas and Castanon to solve the asymmetric assignment problem in which the goal is to match m persons with m out of n objects ($m < n$) while maximizing the total benefit of the match [1]. The algorithm proceeds in alternating rounds of forward and reverse auctions. In the forward stage, people bid for objects and the highest bidders get assigned to the objects. In the reverse stage, objects conceptually bid for people by reducing their prices to attract more persons. It was shown in [1] that this alternation of forward and reverse auctions deals better with price wars than either of its components (only forward or only reverse) and tends to terminate substantially faster than other

approaches.

We can consider the work in [1] as a mean of solving a snapshot of the dynamic distributed task assignment. However, one shortcoming of this approach is that it only deals with unary task requirements (i.e. all tasks require single agent each).

To understand the challenges imposed by dealing with multi-requirement tasks, consider the scenario depicted in Figure 2: a team of three agents are required to service two targets that require three and two agents respectively. A direct application of the scheme in [1] would result in agent 1 and 2 gets assigned to target 1 whereas agent 3 gets assigned to target 2. This pattern would continue indefinitely and would result in no utility gain because of the shape of the targets' utility functions (see Figure 1). To circumvent this problem we propose a dynamic non-linear target utility function. In our scheme, the utility of the target as viewed by a single agent is increased non-linearly with the number of agents assigned to this target (section 3.3). In the previous scenario depicted in Figure 2, this would result in target 1 luring agent 3 to leave target 2 and join agent 1 and 2 in servicing it.

3.1. Overview of the Protocol

We define two main roles in our protocol: the target auctioneer agent and the bidder agent. The former is responsible for running the auction on behalf of the target, while the latter competes with other bidder agents to service this target. It should be noted that the distinction drawn by the above two roles is a functional one rather than being a temporal or existential one. For example, an agent may be the auctioneer for more than one target, play the role of the auctioneer for a given target and bidder for other ones, or even bid for a target for which he is the auctioneer. This is acceptable as we implicitly assume honesty among team members, therefore no agent would give himself a slack for bidding to a target for which it is the auctioneer.

Figure 3 gives an overview of the agent states pertaining to the auction protocols as well as their possible interactions. The auction starts once a new target is detected inside the mission area. The nearest UAV is considered as its auctioneer agent and it announces a new auction. In case of a tie, the highest ID agent is selected. From this point all UAV agents start competing for the new target and the auction runs in rounds each of which is of a predetermined length. During each round, the auctioneer agent receives bids from bidder agents and updates the target price appropriately (section 3.2). At the end of each round, the auctioneer agent evaluates its current state: if it collects the required number of agents for the target, then it announces the result to winner agents which form a winner group to service the target, and this information becomes a common knowledge in the team. Otherwise, the auctioneer agent reduces the price of the current target (reverse step) and propagates this information to the agent team members. The above procedure is repeated until the target is assigned.

¹ In this paper, the term UAV and Agent are used interchangeably.

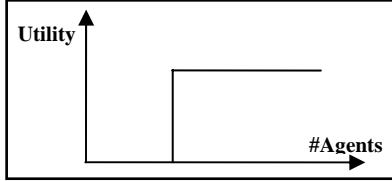


Figure 1 - Step Utility Function

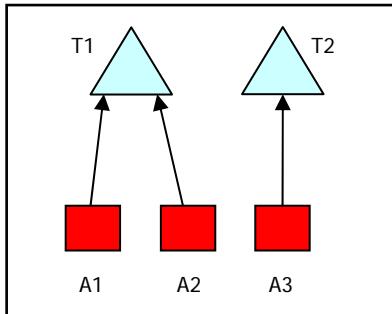


Figure 2 - Multi-requirement Targets Challenges. T1 requires 3 UAVs, and T2 requires 2 UAVs

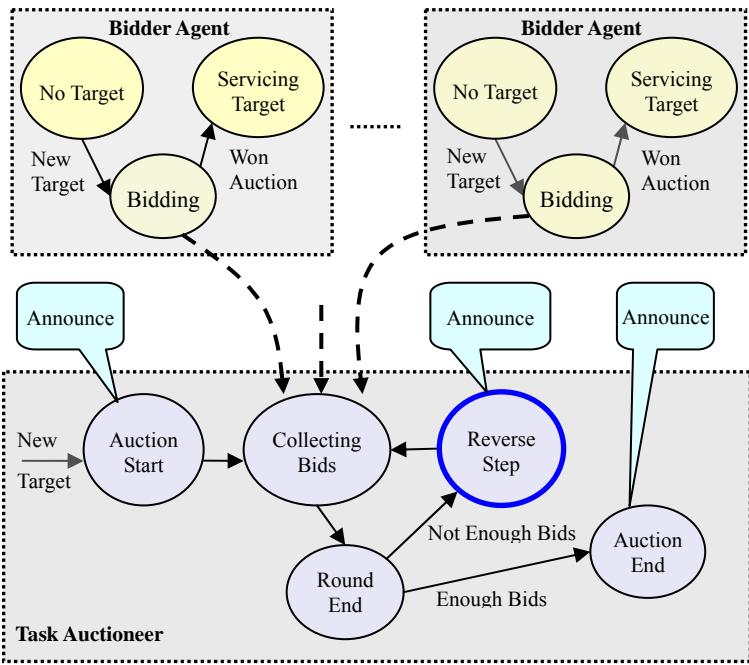


Figure 3 - Inter-roles Interaction in the Forward/Reverse Auction Protocol

It should be noted that the framework allows for more than one auction to exist at the same time. In this case bidder agents need to choose the target to bid for (section 3.3).

At the end of this stage, a near-optimal static assignment is found, and the resulting agent groups are announced by the auctioneers.

3.2. Auctioneer Agent's Policy

The agent who plays the role of the auctioneer for a given target, t , has a very straightforward policy which is parameterized by two variables: ROUND_TIME which is the time period for each forward round and PRICE_REDUCE_RATIO which is the ratio used to update the target price in the reverse step. The auctioneer agent starts by setting the price of the target to zero. During the forward stage, it keeps a decreasing list ($received_bids$) of the received bids. Upon reception of a given bid, it first updates the $received_bids$ and then updates the price of the target ($price_t$) using the Equation 2.

$$price_t = \begin{cases} received_bids[req_t] & \text{if } num(received_bids) \geq req_t \\ \min_i received_bids[i] & \text{otherwise} \end{cases} \quad \dots \dots \dots \quad (2)$$

where req_t is the minimum number of bids to service target t .

Because the auctions are running asynchronously and under the auspice of different auctioneer agents, bidder agents may change their bidding decision and decided to bid for another target (see section 3.3). To do that they send a

BID_RETRACT_REQUEST to the old best target auctioneer agent first. This is because a bid is viewed as a provisional commitment from the bidder to service the target if the auctioneer agent deems him as the winner. This request may be received by the auctioneer agent during the forward round. To respond, the auctioneer agent simply removes the retracted bidder from the $received_bids$ list and updates the target price using Equation 1.

At the end of the round, the auctioneer agent examines the provisional commitments it received. If it has got enough bids, it chooses the best ones based on the target requirement. The auctioneer agent then tries to turn these provisional commitments into a final one by a simple two way handshaking protocol with the winner bidders. This handshaking mechanism is required because the winner bidders might have chosen to retract their provisional commitments, but the requests they sent have not arrived at the auctioneer agent yet. If the auctioneer agent manages to turn sufficient (based on the target requirements) provisional commitments into final ones, then the auction ends and the winners are notified. If the auctioneer agent fails in collecting sufficient final commitments, it proceeds as if it did not acquire the required number of bids. In this case, it updates the target price based on the remaining provisional bidders using Equation 2. After that it reduces the target price by multiplying it with the PRICE_REDUCE_RATIO, which constitutes the reverse step, and broadcasts the new price to start a new forward round. This two round approach is repeated until the target is assigned.

3.3. The Bidding Strategy

The bidder agent's strategy is rather more complex than the auctioneer's one. While the bidder agent is in the bidding

state (see Figure 3), it keeps track of the up-to-date prices of all known non-assigned targets. The bidder agent then needs to make two decisions: 1) which target to bid for and 2) how much to bid for this target. The situation is exacerbated because the information needed to answer these questions (price updates from old auctioneers and new target announcements) arrives asynchronously to the bidder agent. To solve this problem, the bidder agent first answers the above question using Algorithm 1, and whenever the bidder agent receives new information, it re-computes this answer using the same algorithm. If the best target changes, the bidder first retracts its bid from the old target auctioneer agent and then sends a new bid to the new target auctioneer agent.

Algorithm 1 - GetBestTrgetBiddingStructure for agent i

- Calculate the benefit (a_{ij}) of servicing target j as follows

$$a_{ij} = \frac{util_j}{req_j - asn_j} - c_{ij} - price_j$$

- Find j_{i_1}, j_{i_2} as follows:

$$j_{i_1} = \arg \max_{j \in \text{targets}} (a_{ij})$$

$$j_{i_2} = \arg \max_{j \in \text{targets} \setminus \{j_{i_1}\}} (a_{ij})$$

- Bid for task j_{i_1} with value $\pi_{j_{i_1}} = a_{ij_{i_1}} - a_{ij_{i_2}}$

where:

$util_j$ = utility of target j ,

req_j = # of UAV required by target j

asn_j = # of UAV assigned to target j

c_{ij} = cost of agent i to service target j

$price_j$ = current price of target j

Algorithm 1 is very intuitive. First, it computes the first and second maximally beneficial targets. It then bids for the first target with a bid value equal to the difference between the two benefits [1]. In calculating the benefit for target j , the utility of this target is divided by the remaining number of its requirement which constitutes the non-linear utility function explained in section 3.1.

The bidder agent repeats the above procedure until it receives a request from the auctioneer agent to turn its provisional bid into a final one. In this case it needs to check that it is still provisionally committed to this target. If so, it answers positively; otherwise, it answers negatively. Finally, once the bidder agent receives the results of the auction to which it has submitted a bid, it can go in either of two ways: if it wins the bid, the agent starts to work in servicing the target and coordinate with its group members; otherwise it

repeats the above procedure until it gets assigned to a target.

4. SWAPPING: DYNAMIC MAINTENANCE OF AUCTION RESULTS

The auction algorithm results in a near-optimal assignment at the time it was executed [1]. At the end of it, agents are divided into groups each of which is working on servicing a given target. However, as the agents proceed to accomplish this task, the agent states as well as the target states might change in a way that renders this assignment sub-optimal. One way of dealing with this problem is to periodically run the auction algorithm. However, this is a very expensive option and does not make efficient use of the information that the agents have. If we analyze the set of successive optimal assignments, we would discover that they morph into each other seamlessly through a finite set of possible swaps: a change of the assigned targets between two agents. To leverage this observation into an algorithm, we need to design an efficient and robust algorithm that has the following two properties:

- **Efficiency:** All negotiations need to be local between member agents of the two respective groups affected by the swap.
- **Robustness:** Group membership is assumed to be a global knowledge between group members. Therefore, the algorithm should maintain this property across swaps.

Let the current assignment under which the agents are working be S . While each agent is servicing its target, the agent periodically monitors the environment and considers a set of candidate swaps with nearby agents. The agent then examines all these candidate swaps and finds the best swap and its corresponding new assignment S' . The benefit of the swap is computed as the difference between the values of these two assignments using Equation 1. The agent decides to start negotiating the swap with the other group member if the swap benefit is larger than a given threshold. The intuition behind this threshold is twofold: first, it avoids thrashing between groups due to small perturbations of the environment, and second, it provides a way of weighing the benefit of the swap against the resources needed to execute it (the number of messages are needed to maintain intra-group synchronization).

Once a swap is selected for execution by an initiator agent, this agent starts to negotiate the swap with the other member in the thought after group. As usual, synchrony conspires with the distributed nature of the application to only exacerbate the situation. To understand this, consider the situation in which there are two groups each of which has three members. If two different pairs of members have decided to execute the swap at the same time, intra-groups synchronization becomes very hard and expensive to maintain. To prevent this situation from taking place, we stipulate a third requirement:

- **Isolation:** At any given time there can be at most one swap taking place between any two groups.

To achieve these requirements, an agent within each group is assigned to be the group leader. Let a swap take place between an initiator agent and an intended agent. The swap is then proceeds as follows:

1. **Initialization** - The initiator of the swap asks its group leader for permission to start a new swap. This permission is granted as long as there is no other swap taking place.
2. **Asking for a swap** - Upon reception of the swap permission from its own group leader, the initiator agent contacts the member agent involved in the swap in the other group to inform it about executing the swap.
3. **Responding to the swap** - Once the intended agent receives the swap request, it asks for permission from its own group leader. Based on its leader response (grant or decline), it informs the initiator with its decision (accept or decline).
4. **Swap execution** - Once the two parties agreed on the swap, each one informs its peers about the group update.
5. **Finalization** - After the initiator and intended agents inform their peers about the swap, they report to their group leaders that intra-group synchronization has been achieved.

As evident from the above procedure, a swap requires a large number of messages back and forth between group members; therefore the SWAP_THRESHOLD should be adjusted to take this into consideration. Currently, this threshold is treated as a pre-determined parameter to the framework, and its value is set based on a worst case analysis.

5. UAV AGENT ARCHITECTURE

This section provides a function-oriented specification of different components of an UAV agent used in the experimentation and the information flow between them. The emphasis here will be of the reasoning part of the agent.

Figure 4 depicts the UAV agent components as well as information flow between them. The central component of the agent architecture is the world model which is the main memory of the agent. It stores various kind of problem information at different levels of abstractions. The world model is feed with vision data from the communication module, and it provides various retrieval functions to other components of the agent. This allows these components to examine the world around the agent at the most suitable level of details for this component's task.

The coordination module is the brain of the agent where all decisions are taken. It is responsible of sequencing the agent decision making process as well as coordinating the overall team members' actions (team-level action sequencing). The main responsibility of this module is to decide the agent current mode of operation (rooming, bidding, or task

execution), and it also runs the auction framework as detailed before. It also stores partial coordination results on the world model like (which target I am currently committed to which group I am a member of, what is my role in this group, etc.) to be used by other modules.

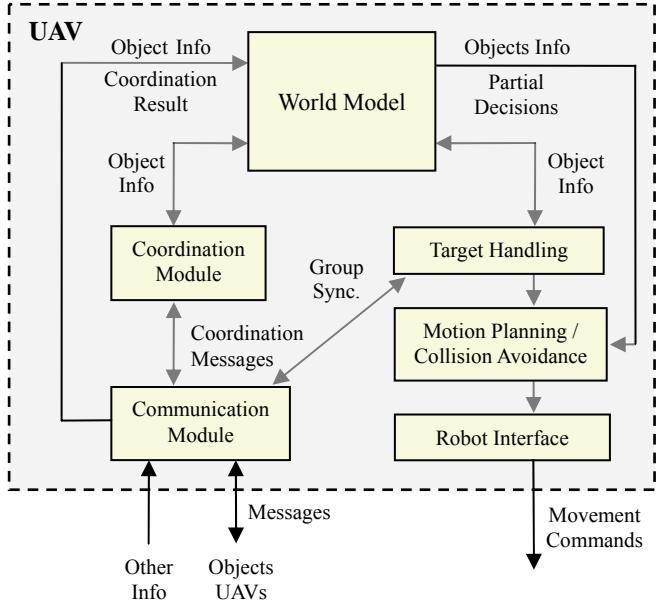


Figure 4 - Internal Architecture of a UAV Agent

The target handling and motion planning modules are responsible for materializing decisions of the coordination module. The responsibilities of these modules are to decide which point of the target to go to and to coordinate with other group members to avoid conflicts. This coordination is done by exchanging path-planning messages between group members through the communication module. Afterwards, the collision avoidance module monitors the world by examining the world model and make sure to avoid/prevent collisions using an A* algorithm [16].

6. EXPERIMENTAL RESULTS

Our main aim is to demonstrate our framework using a set of robot cars. Each car is controlled by an iPAQ and receives localization information from a leader vision server collaborating data from four vision servers, each of which is connected to a ceil-mounted vision camera. A vision server takes images from a camera and processes images searching for color plates mounted over the robot cars, which designate corresponding robots' identification and headings. A leader vision server takes localization information from each vision server, and sends filtered and regulated localization information to iPAQs. Inter-agent communication is achieved via wireless communication between iPAQs. Different cars are used to represent UAVs and targets (see Figure 5). It is quite clear that this hardware setting would make discovering logical errors as well as tracing the agents' behavior very hard. To deal with these issues, we developed a hardware/software shared agent code

architecture that allows us to simulate this hardware setting in software while at the same time guaranteeing interoperability when porting this code to the hardware setting. The main design philosophy of the system is to ease parallel developments and testing. The agent's (UAV/Target) implementation is isolated from the architecture on which the system is running. The system can run in two modes: Simulated Mode or Real Mode.

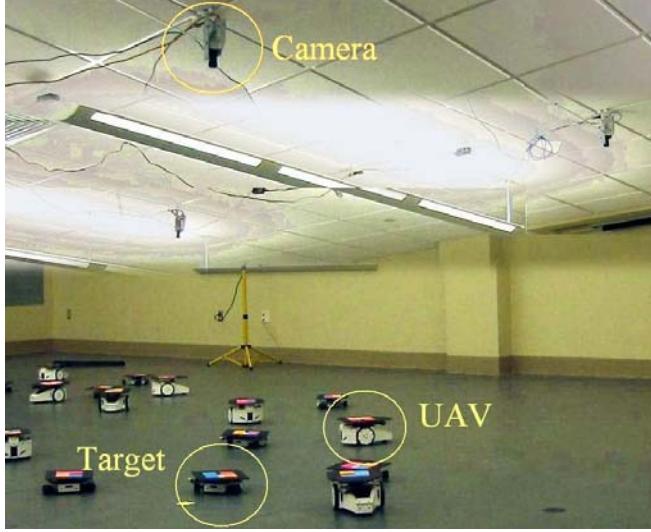


Figure 5 - Experimental Environments

Our experiments are performed with the proposed framework in a 6 UAVs vs. 9 target scenario. The metric we used is the total mission time to service all targets. We ran this mission with different settings for the dynamic auction framework parameters: ROUND_TIME (RT) and PRICE_REDUCE_RATIO (PRR). We also ran experiments with and without the reverse auction stage and with and without swapping. If swapping is disabled, then agents would stick to their initial target assignment even if it becomes sub-optimal later. In all of these experiments, SWAP_THRESHOLD was set to 30 and the swap was considered every 20 seconds (currently, these values are set experimentally).

As evident from the results, swapping and the reverse auction step help reducing the total mission time in all combination of the other parameter settings.

From Table 1, small values of the RT would negatively affect the overall performance. This is largely due to the asynchrony of the application as small RT values would not give the auctioneer agent a chance to receive all bids submitted from bidder agents. This would result in running the auction many times because of not receiving enough bids. The optimal setting for this parameter depends largely on the average message delay. From Table 3, the best PRR setting is 50% which results in the fastest auction termination. We are currently in the process of conducting more experiments to understand the role of this parameter.

Table 1: The effect of auction round time

RT=1 sec		RT=3 sec	
Fwd	Fwd/Rev	Fwd	Fwd/Rev
217 sec	163 sec	175 sec	212 sec

PR=50% with swapping

Table 2: The effect of swapping

With Swapping		Without Swapping	
Fwd	Fwd/Rev	Fwd	Fwd/Rev
217 sec	207 sec	249 sec	163 sec

PR = 50%, RT = 1 sec

Table 3: The effect of the price reduce ratio

Forward Reverse Auction		
PR=10%	PR=30%	PR=50%
207 sec	211 sec	163 sec

RT = 1 sec, with swapping

7. RELATED WORK

The use of economic models in multi-agent coordination in general and task allocation in particular is not new. Various approaches exploited auction-like approaches in both situated [5, 12] and non-situated agents [7, 8]. However, as we detailed before, these approaches lack a formal specification of the DDTA problem. Moreover, they mainly concentrate ([7] slightly depart from this) on how to assign tasks to agents without providing a solid framework to maintain the assignment optimality over time.

The forward/reverse auction was first introduced in [1] to solve asymmetric assignment problem, and therefore it can be regarded as a mean to solve a snapshot of the DDTA problem. Our work here can be viewed as extending it in three directions: first, we allow the algorithm to deal with non-unary task requirements by introducing non-linearity in the task utility function; second, we adapt the initial results of the algorithm using *swapping* to retain its optimality over time; third, we provide a robust implementation for this algorithm that deals with the distributed and asynchronous nature of the DDTA problem. This implantation can be viewed as a relaxed version of the two-phase commit protocols [13].

Swapping is related to conflict management in a team of agents. Several approaches have been proposed to deal with this problem which is based on the shared intention theory [3] in which agents inside the team are assumed to share a common goal. This theory has been materialized in [14, 15] as a set of reusable teamwork heuristic rules that enable

agent teams to act reliably in dynamic situations. However, our work differs from these generic approaches as it was designed specifically to permeate seamlessly with our forward/reverse auction framework.

8. CONCLUSION AND FUTURE WORK

In this paper we have presented a divide and conquer approach for the DDTA problem that deals separately with its two main challenging aspects: combinatorial and dynamicity issues. To deal with the first issue we have adapted a forward reverse auction algorithm which was originally designed for bipartite maximal matching [1] to be suitable for the distributed, asynchronous, multi-requirement aspects of the DDTA problem. We have then proposed to use swapping as a mean to maintain the optimality of the initial auction results over time via chaining of local, communication-inexpensive negotiating steps. We have detailed the application of our approach in a UAV search and rescue domain and reported on early experimentations that prove the promise of our framework.

We plan to run large scale experiments on the same search and rescue domain using our Actor Architecture (AA) [17] which supports up to 10000 agents. We also plan to incorporate learning in our architecture to first tailor the agents' bidding decisions and second to adapt the algorithm parameters to the environment dynamics in order to efficiently utilize the available bandwidth. Moreover, we will also contrast our approach with that in [2] and provide a principled way to contrast them in different settings. Finally, we also plan to investigate theoretically the optimally our multi-requirements extension to the auction algorithm.

ACKNOWLEDGMENT

This research is sponsored by the Defense Advanced Research Projects Agency under contract number F30602-00-2-0586. We would like to thank Hananeh Hajishirazi for implementing part of the collision avoidance and target handling modules. We would like also to thank Soham Mazumdar for early discussion on the forward/reverse auction algorithm.

REFERENCES

- [1] D.P. Bertsekas and D.A. Castanon, "A Forward/reverse Auction Algorithm for Asymmetric Assignment Problems," *Technical Report Lids-P-2159*, MIT, 1993.
- [2] P.J. Modi, H. Jung, M. Tambe, W. Shen, S. Kulkarni, "Dynamic Distributed Resource Allocation: A Distributed Constraint Satisfaction Approach," In *Intelligent Agents VIII Proceedings of the International Workshop on Agents, Theories, Architectures, and Languages (ATAL'01)*, 2001.
- [3] P.R. Cohen and H.J. Levesque, "Confirmation and joint action," In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 951-957, 1991.
- [4] M. Tambe, "Agent Architectures for Flexible, Practical Teamwork," In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, August 1997.
- [5] K. Sycara, A. Pannu, M. Williamson, and D. Zeng, "Distributed Intelligent Agents," *IEEE Expert, Special Issue on Intelligent Systems and their Applications*, 11(6):36-46, December 1996.
- [6] R. Nair, T. Ito, M. Tambe, and S. Marsella, "Task Allocation in the RoboCup Rescue Simulation Domain: A Short Note," In *Proceedings of the International Symposium on RoboCup(RoboCup'01)*, 2001.
- [7] B.P. Gerkey and M.J. Matarić, "Sold!: Auction Methods for Multi-robot Coordination," *IEEE Transactions on Robotics and Automation, Special Issue on Multi-robot Systems*, 18(6):758-768, October 2002.
- [8] P. Caloud, W. Choi, J.-C. Latombe, C. Le Paper, and M. Yim, "Indoor Automation with many Mobile Robots," In *Proceedings of IEE/RSJ International Workshop on Intelligent Robots and Systems*, pages 67-72, 1990.
- [9] M.B. Dias and A. Stentz, "A Market Approach to Multirobot Coordination," *Carnegie Mellon Robotics Institute Technical Report CMU-RI-TR-01-26*, August 2001.
- [10] K.S. Decker, "Environment Centered Analysis and Design of Coordination Mechanisms," *Ph.D. Dissertation*, University of Massachusetts, May 1995.
- [11] R. Davis and R.G. Smith, "Negotiation as a Metaphor for Distributed Problem Solving," *Artificial Intelligence*, 20:63-109, 1983.
- [12] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design, 3rd edition*, Addison-Wesley, 2001.
- [13] G.A. Kaminka and M. Tambe, "Robust Agent Teams via Socially-attentive Monitoring," *Journal of Artificial Intelligence Research*, 12:105-147, 2000.
- [14] M. Tambe, "Agent Architectures for Flexible, Practical Teamwork," In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, August 1997.
- [15] S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach, 2nd edition*, Prentice Hall, Upper Saddle River, NJ, 2003.
- [16] M. Jang, S. Reddy, P. Totic, L. Chen, G. Agha. "An Actor-based Simulation for Studying UAV Coordination," 15th European Simulation Symposium (ESS 2003), pp. 593-601, October 26-29, Delft, The Netherlands, 2003.

Dynamic Agent Allocation for Large-Scale Multi-Agent Applications

Myeong-Wuk Jang and Gul Agha

Department of Computer Science
University of Illinois at Urbana-Champaign,
Urbana IL 61801, USA
`{mjw, agha}@uiuc.edu`

Abstract. Although distributed computing is necessary to execute large-scale multi-agent applications, the distribution of agents is challenging especially when the communication pattern among agents is continuously changing. This paper proposes two dynamic agent allocation mechanisms for large-scale multi-agent applications. The aim of one mechanism is to minimize agent communication cost, while that of the other mechanism is to prevent overloaded computer nodes from negatively affecting overall performance. In this paper, we synthesize these two mechanisms in a multi-agent framework called *Adaptive Actor Architecture* (*AAA*). In *AAA*, each agent platform monitors the workload of its computer node and the communication pattern of agents executing on it. An agent platform periodically reallocates agents according to their communication localities. When an agent platform is overloaded, the agent platform migrates a set of agents, which have more intra-group communication than inter-group or inter-node communication, to a relatively under-loaded agent platform. These agent allocation mechanisms are developed as fully distributed algorithms, and they may move the selected agents as a group. In order to evaluate these mechanisms, preliminary experimental results with large-scale micro UAV (Unmanned Aerial Vehicle) simulations are described.

1 Introduction

Large-scale multi-agent simulations have recently been carried out [8, 12]. These large-scale applications may be executed on a cluster of computers to benefit from distributed computing. When agents participating in a large-scale application communicate intensively with each other, the distribution of agents on the cluster may significantly affect the performance of multi-agent systems: overloaded computer nodes may become the bottleneck for concurrent execution, or inter-node communication may considerably delay computation.

Many load balancing and task assignment algorithms have been developed to assign tasks on distributed computer nodes [13]. These algorithms mainly use information about the amount of computation and the inter-process communication cost; a task requires a small amount of computational time to finish, and the

communication cost of tasks is known *a priori*. However, in many multi-agent applications, agents do not cease from execution until their system finishes the entire operation [5]. Furthermore, since the communication pattern among co-operative agents is continuously changing during execution, it may be infeasible to estimate the inter-agent communication cost for a certain time period. Therefore, task-based load balancing algorithms may not be applicable to multi-agent applications.

This paper proposes two agent allocation mechanisms to handle the dynamic change of the communication pattern of agents participating in a large-scale multi-agent application and to move agents on overloaded computer nodes to relatively underloaded computer nodes. *Adaptive Actor Architecture* (AAA), the extended multi-agent framework of *Actor Architecture* [9], monitors the status of computer nodes and the communication pattern of agents, and migrates agents to collocate intensively communicating agents on a single computer node. In order to move agents to another computer node, an agent platform on a single node manages virtual agent groups whose member agents have more intra-group communication than inter-group or inter-node communication. In order to evaluate our approach, large-scale micro UAV (Unmanned Aerial Vehicle) simulations including 10,000 agents were tested.

This paper is organized as follows. Section 2 introduces the overall architecture of our agent system. Section 3 explains in details two dynamic agent allocation mechanisms of our agent system. Section 4 shows the preliminary experimental results to evaluate these allocation mechanisms, and Section 5 describes related work. Finally, Section 6 concludes this paper with our future work.

2 Adaptive Actor Architecture

AAA provides a light-weight implementation of agents as active objects or actors [1]; agents in AAA are implemented as threads instead of processes, and they communicate using object messages instead of string messages. The actor model provides the fundamental behavior for a variety of agents; they are social and reactive, but they are not explicitly required to be “autonomous” in the sense of being proactive [16]. However, autonomous actors may be implemented in AAA, and many of the applications used in our experimental studies require proactive actors. Although the term agent has been used to mean proactive actors, for our purposes the distinction is not critical. In this paper, we use the terms ‘agent’ and ‘actor’ as synonyms.

Adaptive Actor Architecture consists of two main parts:

- *AAA platforms* which provide the system environment in which agents exist and interact with other agents. In order to execute agents, each computer node must have one AAA platform. AAA platforms provide agent state management, agent communication, agent migration, agent monitoring, and middle agent services.

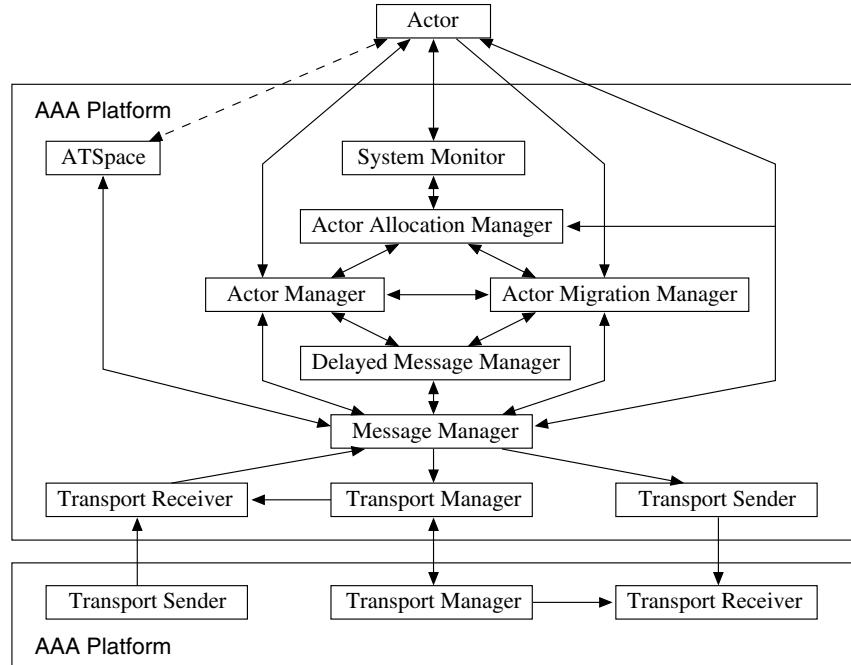


Fig. 1. Architecture of an AAA Platform

- *Actor library* which is a set of APIs that facilitate the development of agents on the AAA platforms by providing the user with a high level abstraction of service primitives. At execution time, the actor library works as the interface between agents and their respective AAA platforms.

An AAA platform consists of ten components (see Fig. 1): Message Manager, Transport Manager, Transport Sender, Transport Receiver, Delayed Message Manager, Actor Manager, Actor Migration Manager, Actor Allocation Manager, System Monitor, and ATSpace.

The *Message Manager* (MM) handles message passing between agents. Every message passes through at least one Message Manager. If the receiver agent of a message exists on the same AAA platform as the sender agent, the MM of the platform directly delivers the message to the receiver agent. However, if the receiver agent is not on the same AAA platform, this MM delivers the message to the MM of the platform where the receiver currently resides, and finally the MM delivers the message to the receiver. The *Transport Manager* (TM) maintains a public port for message passing between different AAA platforms. When a sender agent sends a message to a receiver agent on a different AAA platform, the *Transport Sender* (TS) residing on the same platform as the sender receives the message from the MM of the sender agent and delivers it to the *Transport Receiver* (TR) on the AAA platform of the receiver. If there is no

built-in connection between these two AAA platforms, the TS contacts the TM of the AAA platform of the receiver agent to open a connection so that the TM creates a TR for the new connection. Finally, the TR receives the message and delivers it to the MM on the same platform.

The *Delayed Message Manager* (DMM) temporarily holds messages for mobile agents while they are moving from their AAA platforms to other AAA platforms. The *Actor Manager* (AM) manages states of the agents that are currently executing and the locations of the mobile agents created on the AAA platform. The *Actor Migration Manager* (AMM) manages agent migration.

The *System Monitor* (SM) periodically checks the workload of its computer node and an *Actor Allocation Manager* (AAM) analyzes the communication pattern of agents. With the collected information, the AAM makes decisions for either agents or agent groups to deliver to other AAA platforms with the help of the Actor Migration Manager. The AAM negotiates with other AAMs to check the feasibility of migrations before starting agent migration.

The *ATSpace* provides middle agent services, such as matchmaking and brokering services. Unlike other system components, the ATSpace is implemented as an agent. Therefore, any agent can create an ATSpace, and hence, an AAA platform may have more than one ATSpaces. The ATSpace created by an AAA platform is called the *default ATSpace* of the platform, and all agents can obtain the agent names of default ATSpaces. Once an agent has the name of an ATSpace, the agent may send the ATSpace messages in order to use its services, and the messages are delivered through the Message Manager.

3 Dynamic Agent Allocation

In order to develop large-scale distributed multi-agent applications, the multi-agent systems must be scalable. This scalability may be achieved if the application or the infrastructure does not include centralized components which can become a bottleneck. Moreover, the scalability requires relatively balanced workload on computer nodes. Otherwise, the slowest node may become a bottleneck. However, balancing the workload between computer nodes requires significant overhead: the relevant global state information needs to be gathered, and agents have to transferred sufficiently frequently between computer nodes. Therefore, when the number of computer nodes and/or the number of agents is very large, load balancing is difficult to achieve. AAA uses the *load sharing approach* in which agents on an overloaded agent platform are moved to other underloaded agent platforms, but balanced workload among computer nodes is not required.

The third important factor for the scalability of multi-agent systems is the communication overhead. When agents on separate computer nodes communicate intensively with each other, this factor may significantly affect the performance of multi-agent systems. Even though the speed of local networks has increased considerably, the intra-node communication speed for agent message passing is much faster than inter-node communication. Therefore, if we can collocate together agents which communicate intensively with each other, com-

munication time significantly decreases. It is not generally feasible for a user to distribute agents based on their communication pattern, because the communication pattern among agents may change over time in unpredictable ways. Therefore, agents should be reallocated dynamically according to their communication patterns, and this procedure should be managed by a middleware system, such as agent platforms. Each agent platform in AAA monitors the status of its computer node and the communication pattern of agents on it, and the platform dynamically reallocates agents according to the information gathered.

3.1 Agent Allocation for Communication Locality

An agent allocation mechanism used in AAA handles the dynamic change of the communication pattern among agents. This mechanism consists of four phases: *monitoring*, *agent allocating*, *negotiation*, and *agent migration* (see Fig 2).

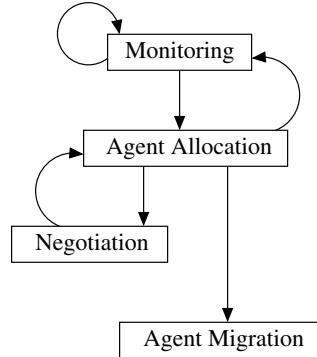


Fig. 2. Four Phases for Basic Dynamic Agent Allocation

Monitoring Phase The Actor Allocation Manager checks the communication pattern of agents under the support of the Message Manager. The Actor Allocation Manager makes a log with information about both the sender agent and the agent platform of the receiver agent of each message. Therefore, each agent element in the Actor Allocation Manager has variables representing all agent platforms communicating with this agent; M_{ij} is the number of messages sent from agent i to agent platform j .

Periodically or when requested by a system agent, the Actor Allocation Manager updates the communication pattern between agents and agent platforms with the following equation:

$$C_{ij}(t) = \alpha \left(\frac{M_{ij}(t)}{\sum_k M_{ik}(t)} \right) + (1 - \alpha) C_{ij}(t - 1)$$

where $C_{ij}(t)$ is the communication dependency between agent i and agent platform j at the time step t ; $M_{ij}(t)$ is the number of messages sent from agent i to agent platform j during the t -th time step; and α is a coefficient for the relative importance between recent information and old information.

For analyzing the communication pattern of agents, agents in AAA are classified into two types: *stationary* and *movable*. Any agent in AAA can move itself according to its decision, even though it is either stationary or movable. However, the Actor Allocation Manager does not consider stationary agents as candidates for agent allocation; an agent platform can migrate only movable agents. These types of agents are initially decided by agent programmers, and may be changed during execution by the agents, but not by agent platforms.

Agent Allocation Phase After a certain number of repeated monitoring phases, the Actor Allocation Manager computes the communication dependency ratio of an agent between its current agent platform and another agent platform:

$$R_{ij} = \frac{C_{ij}}{C_{in}}, \quad j \neq n$$

where R_{ij} is the communication dependency ratio of agent i between its current agent platform n and agent platform j .

When the maximum ratio of an agent is larger than a predefined threshold, the Actor Allocation Manager assigns this agent to a virtual agent group that represents the remote agent platform:

$$\max(R_{ij}) > \theta \rightarrow a_i \in G_j$$

where θ is the threshold for agent migration, a_i represents agent i , and G_j means agent group j .

When the Actor Allocation Manager has checked all agents and assigned some of them to agent groups, the Actor Allocation Manager starts the negotiation phase. After the agent allocation phase, information about the communication dependency of agents is reset.

Negotiation Phase Before an agent platform migrates the agents that are in an agent group to another agent platform, the Actor Allocation Manager of the sender agent platform communicates with that of the destination agent platform to check its current status. If the destination agent platform has enough space and available computational resources for new agents, its Actor Allocation Manager accepts the request for the agent group migration. Otherwise, the destination agent platform sends the number of agents that it can accept. The granularity of this negotiation between agent platforms is an agent. When the Actor Allocation Manager receives a reply from the destination agent platform, the Actor Allocation Manager sends as many agents to the destination agent platform as the number of agents recorded in the reply message. When the number in the reply message is less than the number of agents in the virtual

group, the agents to be migrated are selected according to their communication dependency ratios.

Agent Migration Phase When the destination agent platform can accept new agents, the Actor Allocation Manager of the sender agent platform initiates the migration of agents in the selected agent groups. After the current operation of a selected agent finishes, the Actor Migration Manager moves the agent to the destination agent platform decided by the Actor Allocation Manager. After the agent is migrated, the agent may restart its remaining operations.

3.2 Agent Allocation for Load Sharing

With the previous agent allocation mechanism, AAA handles the dynamic change of the communication pattern of agents. However, this mechanism may increase the workload of certain agent platforms. Therefore, our agent allocation has been extended. When an agent platform is overloaded, the System Monitor detects this and activates the agent reallocation procedure. Since agents had been assigned to their current agent platforms according to their communication pattern, choosing agents randomly to migrate to underloaded agent platforms might result in moving them back to their original agent platforms by the Actor Allocation Managers of their new agent platforms. This is because the moved agents may still have a high communication with their previous agent platform. This agent allocation mechanism consists of five phases: *monitoring*, *agent grouping*, *group allocation*, *negotiation*, and *agent migration* (see Fig 3).

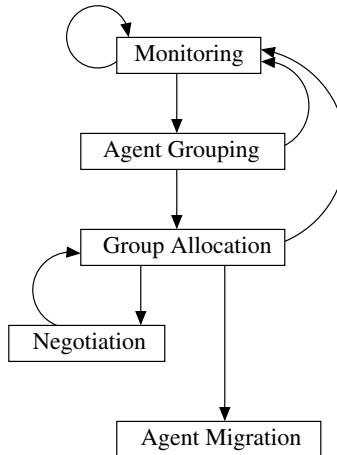


Fig. 3. Five Phases for Extended Dynamic Agent Allocation

Monitoring Phase In the second agent allocation mechanism, the System Monitor periodically checks the state of its agent platform; the System Monitor gathers information about the current processor usage and the memory usage of its computer node. When the System Monitor decides that its agent platform is overloaded, it activates the agent allocation procedure. When the Actor Allocation Manager is notified by the System Monitor, it starts monitoring the local communication pattern among agents and classifies them to agent groups. If an agent belonged to an agent group, it is assigned to this agent group; if an agent did not belong to any agent group, it is randomly assigned to an agent group. The number of agent groups that exist on an agent platform is predefined.

For checking the communication pattern of agents, the Actor Allocation Manager makes a log with information about the sender agent, the agent platform of the receiver agent, and the agent group of the receiver agent of each message. In addition to the number M_{ij} of messages sent from agent i to agent platform j , the number m_{ik} of messages sent from agent i to agent group k is updated when the receiver agent exists on the same agent platform. The summation of all m variables of an agent is equal to the number of messages sent by the agent to its current agent platform: $\sum_k m_{ik} = M_{in}$ where the index of the current agent platform is n .

After a predetermined time interval, or in response to a request from a system agent, the Actor Allocation Manager updates the communication pattern between agents and agent groups on the same agent platform with the following equation:

$$c_{ij}(t) = \beta \left(\frac{m_{ij}(t)}{\sum_k m_{ik}(t)} \right) + (1 - \beta)c_{ij}(t - 1)$$

where $c_{ij}(t)$ is the communication dependency between agent i and agent group j at the time step t ; $m_{ij}(t)$ is the number of messages sent from agent i to agents in agent group j during the t -th time step; and β is a coefficient for deciding the relative importance between recent information and old information.

Agent Grouping Phase After a certain number of repeated monitoring phases, each agent i is assigned to an agent group whose index is decided by $\arg \max_j (c_{ij}(t))$; this group has the maximum value of the communication localities $c_{ij}(t)$ of agent i . Since the initial group assignment of agents may not be well organized, the monitoring and agent grouping phases are repeated.

After each agent grouping phase, information about the communication dependency of agents is reset. During the agent grouping phase, the number of agent groups can be changed. When two groups have much smaller populations than others, these two groups may be merged into one group. When one group has a much larger population than others, the agent group may be split into two groups. The minimum number and maximum number of agent groups are predefined.

Group Allocation Phase After a certain number of repeated monitoring and agent grouping phases, the Actor Allocation Manager makes a decision to move an agent group to another agent platform. The group selection is based on the communication dependency between agent groups and agent platforms; the communication dependency D_{ij} between agent group i and agent platform j is decided by the summation of the communication dependency between agents in the agent group and the agent platform:

$$D_{ij} = \sum_k C_{kj}(t) \quad \text{where } a_k \in A_i$$

where A_i represents the agent group i , and a_k is a member agent of the agent group A_i .

The agent group which has the least dependency to the current agent platform is selected; the index of the group is decided by the following equation:

$$\arg \max_j \left(\frac{\sum_{j,j \neq n} D_{ij}}{D_{in}} \right)$$

where n is the index of the current agent platform. The destination agent platform of the selected agent group i is decided by the communication dependency between the agent group and agent platforms; the index of the destination platform is $\arg \max_j (D_{ij})$.

Negotiation Phase If one agent group and its destination agent platform are decided, the Actor Allocation Manager communicates with that of the destination agent platform. If the destination agent platform accepts all agents in the group, the Actor Allocation Manager of the sender agent platform starts the migration phase. Otherwise, this Actor Allocation Manager communicates with that of the second best destination platform until it finds an available destination agent platform or checks the possibility of all other agent platforms.

This phase of the second agent allocation mechanism is similar to that of the previous agent allocation mechanism, but there are some differences. One important difference between these two negotiation phases is the granularity of negotiation. If the destination agent platform has enough space and available computation power for all agents in the selected agent group, the Actor Allocation Manager of the destination agent platform accepts the request for the agent group migration. Otherwise, the destination agent platform refuses the request. The granularity of this negotiation between agent platforms is an agent group; the destination agent platform cannot accept part of an agent group.

Agent Migration Phase When the sender agent platform receives the acceptance reply from the destination agent platform, the Actor Allocation Manager of the sender agent platform initiates the migration of agents in the selected agent group. The procedure for the following phase in the second agent allocation mechanism is the same as that of the previous agent allocation mechanism.

3.3 Characteristics

Transparent Distributed Algorithm These agent allocation mechanisms are developed as fully distributed algorithms; each agent platform independently performs its agent allocation mechanism according to information about its workload and the communication pattern of agents on it. There are no centralized components to manage the overall procedure of agent allocation. These mechanisms are transparent to multi-agent applications. The only requirement for application developers is to declare candidate agents for agent allocation as movable.

Load Balancing vs. Load Sharing The second agent allocation mechanism is not a load balancing mechanism but a load sharing mechanism; it does not try to balance the workload of computer nodes participating in an application. The goal of our multi-agent system is to reduce the turnaround time of applications with optimized agent allocation. Therefore, only overloaded agent platforms perform the second agent distribution mechanism, and agents are moved from overloaded agent platforms to underloaded agent platforms.

Individual Agent-based Allocation vs. Agent Group-based Allocation With the agent group-based allocation mechanism, some communication locality problems may be solved. First, when two agents on the same agent platform communicate intensively with each other but not with other agents on the same platform, these agents may continuously stay on the current agent platform even though they have a large amount of communication with agents on another agent platform. If these two agents can move together to the remote agent platform, the overall performance can be improved. However, an individual agent-based allocation mechanism does not handle this situation. Second, individual agent allocation may require much platform-level message passing among agent platforms for the negotiation. For example, in order to send agents to other agent platforms, agent platforms should negotiate with each other to avoid sending too many agents to a certain agent platform, thus overloading the agent platform. However, if an agent platform sends a set of agents at one time, the agent platforms may reduce negotiation messages and negotiation time.

Stop-and-Repartitioning vs. Implicit Agent Allocation Some object reallocation systems require the global synchronization. This kind approach is called the stop-and-repartitioning [2]. Our agent distribution mechanisms are executed in parallel with applications. The monitoring and agent allocation phases do not interrupt the execution of application agents.

Size of a Time Step In the monitoring phase, the size of each time step may be fixed. However, this step size may be adjusted by an agent application. For example, in multi-agent based simulations, this size may be the same as the size

of a simulation time step. Thus, the size of time steps may be flexible according to the workload of each simulation step and the processor power. To use dynamic step size, our agent system has a *reflective mechanism*; agents in applications are affected by multi-agent platform services, and the services of the multi-agent platform may be controlled by agents in applications.

4 Experimental Results

For the purpose of evaluation, we provide experimental results related to micro UAV (Unmanned Aerial Vehicle) simulations. These simulations include from 2,000 to 10,000 agents; half of them are UAVs, and the others are targets. Micro UAVs perform a surveillance mission on a mission area to detect and serve moving targets. During the mission time, these UAVs communicate with their neighboring UAVs to perform the mission together. The size of a simulation time step is one half second, and the total simulation time is around 37 minutes. The runtime of each simulation depends on the number of agents and the collaboration policy among agents. For these experiments, we have used four computers (3.4 GHz Intel CPU and 2 GB main memory) with a Giga-bit switch.

For UAV simulations, the *agent-environment interaction* model has been used [3]; all UAVs and targets are implemented as intelligent agents, and the navigation space and radar censors of all UAVs are implemented as environment agents.

To remove centralized components in distributed computing, each environment agent on a single computer node takes charge of a certain navigation area. UAVs communicate directly with each other and indirectly with neighboring UAVs and targets through environment agents. Environment agents provide application agent-oriented brokering services with the ATSpace [10]. During simulation, UAVs and targets move from one divided area to another, and UAVs and targets communicate intensively either directly or indirectly.

Fig. 4 depicts the difference of runtimes in two cases: dynamic agent allocation, and static agent allocation. Fig. 5 shows the ratio of runtimes in both cases. These two figures show the potential performance benefit of dynamic agent allocation. In our particular example, as the number of agents is increased, the ratio also generally increases. With 10,000 agents, the simulation using the dynamic agent allocation is more than five times faster than the simulation with a static agent allocation.

5 Related Work

The mechanisms used in dynamic load balancing may be compared to those in AAA. *Zoltan* [7], *PREMA/ILB* [2], and *Charm++* [4] support dynamic load balancing with object migration. Zoltan uses a loosely coupled approach between applications and load balancing algorithms using an object-oriented callback function interface [7]. However, this library-based load balancing approach depends on information given by applications, and applications activate object

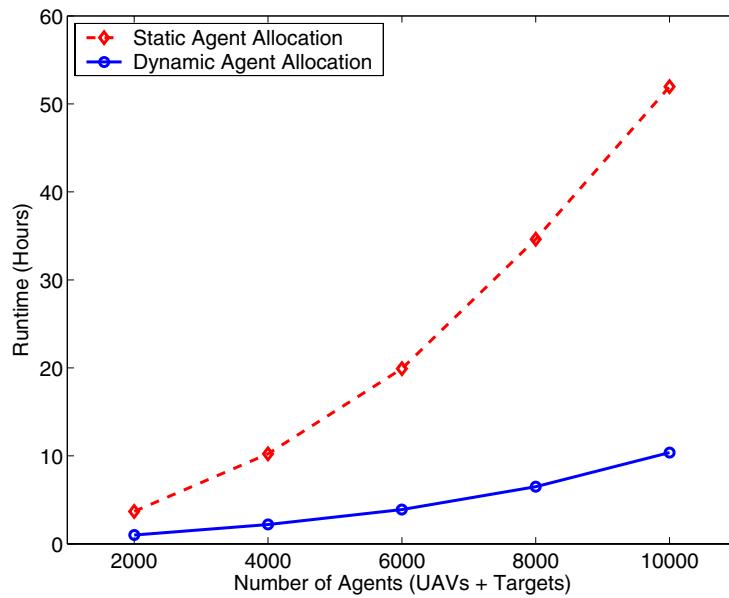


Fig. 4. Runtime for Static and Dynamic Agent Allocation

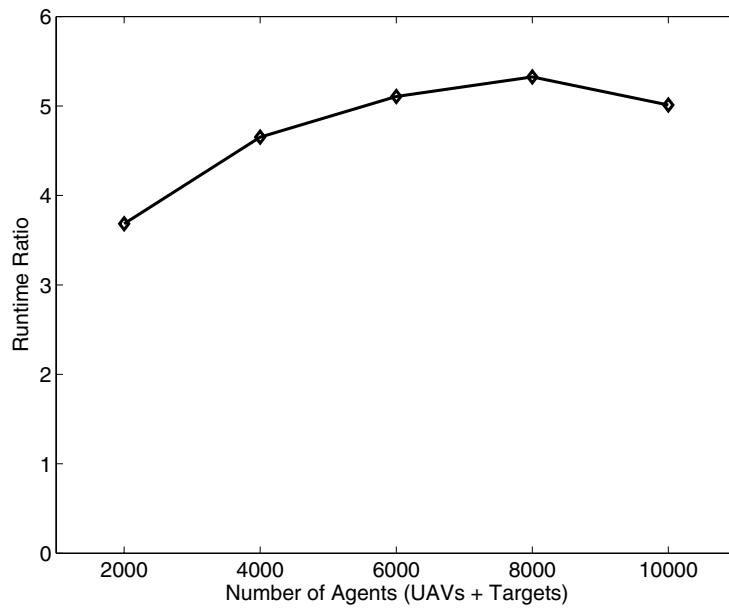


Fig. 5. Runtime Ratio of Static-to-Dynamic Agent Allocation

decomposition. Therefore, without developers' thorough analysis about applications, the change of dynamic access patterns of objects may not correctly be detected, and object decomposition may not be performed at the proper time. The ILB of PREMA also interacts with objects using callback routines to collect information to be used for the load balancing decision making, and to pack and unpack objects [2]. Charm++ uses the *Converse* runtime system to maintain message passing among objects, and hence, the runtime system may collect information to analyze communication dependencies among objects [4]. However, this system also requires callback methods for packing and unpacking objects as others do. In AAA, the Actor Allocation Manager does not interact with agents, but it receives information from the Message Manager and the System Monitor to analyze the communication patterns of agents and the workload of its agent platform. Also, developers do not need to define any callback method for load balancing.

J-Orchestra [15], *Addistant* [14], and *JavaParty* [11] are automatic application partitioning systems for Java applications. They transform input Java applications into distributed applications using a bytecode rewriting technique. They can migrate Java objects to take advantage of locality. However, they differ from AAA in two ways. First, while they move objects to take advantage of data locality, AAA migrates agents to take advantage of communication locality. Second, the access pattern of an object differs from the communication pattern of an agent. For example, although a data object may be moved whenever it is accessed by other objects on different platforms, an agent cannot be migrated whenever it communicates with other agents on different platforms. This is because an object is accessed by another single object, but an agent communicates with other multiple agents at the same time.

The *Comet* algorithm assigns agents to computer nodes according to their credit [5]. The credit of an agent is decided by its computation load, intra-communication load, and inter-communication load. Chow and Kwok have emphasized the importance of the relationship between intra-communication and inter-communication of each agent. However, there are some important differences. The authors' system includes a centralized component to make decisions for agent assignment, and their experiments include a small number of agents, i.e., 120 agents. AAA uses fully distributed algorithm, and experiments include 10,000 agents. Because of the large number of agents, the Actor Allocation Manager cannot analyze the communication dependency among all individual agents, but only that between agents and agent platforms and that between agent groups and agent platforms.

The *IO* of *SALSA* [6] provides various load balancing mechanisms for multi-agent applications. The IO also analyzes the communication pattern among individual agents. Therefore, it may not be applied to large-scale multi-agent applications because of the large computational overhead.

6 Conclusion and Future Work

This paper has explained two dynamic agent allocation mechanisms used in our multi-agent middleware called Adaptive Actor Architecture; these agent allocation mechanisms distributes agents according to their communication localities and the workload of computer nodes participating in large-scale multi-agent applications. The main contribution of this paper is to provide agent allocation mechanisms to handle a large number of agents which communicate intensively with each other and change their communication localities. Because of the large number of agents, these agent allocation mechanisms focus on the communication dependencies between agents and agent platforms and the dependencies between agent groups and agent platforms, instead of the communication dependencies among individual agents. Our experimental results show that micro UAV simulations using the dynamic agent allocation are approximately five times faster than those with a static agent allocation.

Our experiments suggest that increased load does not necessarily result in a decrease in the performance of multi-agent applications. If agents are properly located according to their communication pattern, the processor usage of their agent platforms is quite high. Adding more computer nodes can increase the turnaround time of the entire computation; when the number of agent platforms for an application exceeds a certain limit, the inter-node communication cost becomes larger than the benefit of distributed computing. Therefore, we plan to develop algorithms to determine the appropriate number of agent platforms for a large-scale multi-agent application.

Acknowledgements

This research is sponsored by the Defense Advanced Research Projects Agency under contract number F30602-00-2-0586.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
2. K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali. A Load Balancing Framework for Adaptive and Asynchronous Applications. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):183–192, February 2004.
3. M. Bouzid, V. Chevrier, S. Vialle, and F. Charpillet. Parallel Simulation of a Stochastic Agent/Environment Interaction. *Integrated Computer-Aided Engineering*, 8(3):189–203, 2001.
4. R.K. Brunner and L.V. Kalé. Adaptive to Load on Workstation Clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112, February 1999.
5. K. Chow and Y. Kwok. On Load Balancing for Distributed Multiagent Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(8):787–801, August 2002.

6. T. Desell, K. El Maghraoui, and C. Varela. Load Balancing of Autonomous Actors over Dynamic Networks. In *Hawaii International Conference on System Sciences HICSS-37 Software Technology Track*, Hawaii, January 2004.
7. K. Devine, B. Hendrickson, E. Boman, M. St. Jhon, and C. Vaughan. Design of Dynamic Load-Balancing Tools for Parallel Applications. In *Proceedings of the International Conference on Supercomputing*, pages 110–118, Santa Fe, 2000.
8. L. Gasser and K. Kakugawa. MACE3J: Fast Flexible Distributed Simulation of Large, Large-Grain Multi-Agent Systems. In *Proceedings of the First International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, pages 745–752, Bologna, Italy, July 2002.
9. M. Jang and G. Agha. On Efficient Communication and Service Agent Discovery in Multi-agent Systems. In *Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS '04)*, pages 27–33, Edinburgh, Scotland, May 24-25 2004.
10. M. Jang, A. Abdel Momen, and G. Agha. ATSpace: A Middle Agent to Support Application-Oriented Matchmaking and Brokering Services. In *IEEE/WIC/ACM IAT(Intelligent Agent Technology)-2004*, pages 393–396, Beijing, China, September 20-24 2004.
11. M. Philippse and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.
12. K. Popov, V. Vlassov, M. Rafea, F. Holmgren, P. Brand, and S. Haridi. Parallel Agent-Based Simulation on a Cluster of Workstations. *Parallel Processing Letters*, 13(4):629–641, 2003.
13. P.K. Sinha. Chapter 7. Resource Management. In *Distributed Operating Systems*, pages 347–380. IEEE Press, 1997.
14. M. Tatubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of 'Legacy' Java Software. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 236–255, Budapest, June 2001.
15. E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, Malaga, June 2002. <http://j-orchestra.org/>.
16. M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, Ltd, 2002.

Appendix X:

Maximal Clique Based Distributed Group Formation For Task Allocation in Large-Scale Multi-Agent Systems

Predrag T. Tosic and Gul A. Agha

Open Systems Laboratory, Department of Computer Science

University of Illinois at Urbana-Champaign

Mailing address: Siebel Center for Computer Science,

201 N. Goodwin Ave., Urbana, IL 61801, USA

p-tosic@cs.uiuc.edu, agha@cs.uiuc.edu

Abstract. *We present a fully distributed algorithm for group or coalition formation among autonomous agents. The algorithm is based on two main ideas. One is a distributed computation of maximal cliques (of up to pre-specified size) in the underlying graph that captures the interconnection topology of the agents. Hence, given the current configuration of the agents, the groups that are formed are characterized by a high degree of connectivity, and therefore high fault tolerance with respect to node and link failures. The second idea is that each agent chooses its most preferable coalition based on how highly the agent values each such coalition in terms of the coalition members' combined resources or capabilities. Coalitions with sufficient resources for fulfilling particular highly desirable task(s) are preferable to coalitions with either insufficient resources, or with resources that suffice only for completing less valuable tasks. We envision variants of our distributed algorithm herein to prove themselves useful coordination subroutines in many massively multi-agent system applications where the agents may repeatedly need to form temporary groups or coalitions of modest sizes in an efficient, online and fully distributed manner.*

Keywords: *distributed algorithms, large-scale multi-agent systems, distributed group formation, agent coalitions*

1 Introduction and Motivation

Autonomous agents and *multi-agent systems (MAS)* [1, 4, 29, 30] are characterized, among other properties, by (i) a considerable degree of autonomy of individual computing entities or processes (agents) and, at the same time, the fact that (ii) each agent has a local, that is, in general, incomplete and imperfect “picture of the world”. Since in MAS there is either no central control, or at best only a limited central control, and the individual agents have to both think and act locally, genuinely distributed algorithms are needed for the agents to effectively coordinate with one another.

MAS pose a number of challenges to a distributed algorithm designer. Many of the major challenges are related to various aspects of the agent coordination problem (e.g., [1, 29]). In order to be able to effectively coordinate, agents need to be able to *reach consensus (or agreement)* on various matters of common interest. Two particularly prominent distributed consensus problems are those of *leader election* (e.g., [8, 22]) and *group formation*. Group or coalition formation is an important issue in distributed systems in general [8], and MAS in particular [33]. Given a collection of communicating agents, the goal in distributed group formation is that these agents, based on their local knowledge only, decide how to effectively split up into several groups, so that each agent knows which group(s) it belongs to.

There are several critical issues that a MAS designer needs to address in the context of (distributed) group formation. First, what is the right notion of a group in a given setting? Second, a distributed group formation mechanism - that is, a distributed algorithm that enables agents to effectively form groups or coalitions - needs to be provided. Third, groups and each agent's knowledge about its group membership need to be maintained and, when needed, appropriately updated. Fourth, are the groups to be allowed to overlap, so that an agent may simultaneously belong to two or more groups? These and other challenges related to autonomous agents forming coalitions have been extensively studied in the literature on

multi-agent systems, e.g., [9, 11, 14, 18, 19, 33]. They have also arisen in our own recent and ongoing work on parametric models and a scalable simulation of large scale ($10^3 - 10^4$ agents) ensembles of autonomous unmanned vehicles on a multi-task mission [6, 7, 23, 24].

Herein, we restrict our attention to the second issue above. We propose a particular mechanism (distributed algorithm) for an effective coalition formation that ensembles of autonomous agents can use as one of their basic coordination subroutines. A need for a dynamic, fully distributed, efficient and online group formation may arise due to a number of different factors, such as the geographical dispersion of the agents, heterogeneity of tasks and their resource requirements, heterogeneity of agents' capabilities, and so on [24]. While for small- and medium-scale systems of robots or unmanned vehicles a fully or partially centralized approach to group formation and maintenance may be feasible or even optimal, large scale systems (with the number of agents of orders of magnitude $10^3 - 10^4$ or higher) appear to necessitate a fully distributed approach. That is, the agents need to be able to self-organize into coalitions, and quickly reach a consensus on who is forming a coalition with whom in a fully decentralized manner [25].

2 Group or Coalition Formation in Multi-Agent Systems

Large ensembles of autonomous agents provide an important class of examples where the agents' capability to coordinate and, in particular, to self-organize into groups or coalitions, is often of utmost importance for such systems to be able to accomplish their tasks.

One can distinguish between two general, broad classes of such autonomous agents. One is the class of agents deployed in the context of *distributed problem solving*. The agents encountered in distributed problem solving (DPS) typically share their goal(s). For instance, DPS agents most often have a joint utility function that they all wish to maximize as a team, without any regard to (or perhaps even a notion of) individual payoffs. This joint utility or, more generally, the goal or set of goals assigned to DPS agents, is usually provided by their designer. However, it may not be always feasible - or even possible - that the designer always explicitly specify, for instance, how are the agents to divide-and-conquer their tasks and resources, how are they to form groups and elect leaders of those groups, etc. Due to scalability, incomplete *a priori* knowledge of the environments these agents may encounter, and possibly other considerations, instead of "hard-wiring" into his DPS agents explicitly how are the agents to be coordinated, the system designer may choose only to enable the autonomous agents with the basic coordination primitives, and leave to the agents to self-organize and coordinate as the situation may demand. Hence, in many situations the DPS agents may be required to be able to effectively form groups or coalitions in a fully distributed manner.

The second basic type of agents, the *self-interested agents*, are a kind of agents that do not share their goals (and, indeed, need not share their designer). In contrast to the DPS agents, each self-interested agent has its own agenda (e.g., an individual utility function it is striving to maximize), and no altruistic incentives to cooperate with other agents. However, even such self-interested, goal-driven or individual-utility-driven agents, while in essence selfish, may nonetheless still need to cooperatively coordinate and collaborate with each other in many situations. One class of examples are those agents (such as, e.g., autonomous unmanned vehicles) that, if they do not coordinate in order to resolve possible conflicts, they risk mutual annihilation. Another class of examples are the agents with bounded resources: individually, an agent may lack resources to accomplish any of its desired tasks - yet, if this agent forms a coalition with one or more other agents, the combined resources and joint effort of all agents in such a coalition may provide utility benefits to everyone involved. An example are heterogeneous types of agents deployed to a rescue mission in a disaster area: while there is an overarching, global goal at the system level, each agent typically also has its local, individual notion of task(s) and its own goal(s).

For these reasons, group and coalition formation are of considerable interest for many different kinds of autonomous agents and multi-agent systems, and, among other, even in those multi-agent systems where the agents do not share a global utility function, and where each agent generally acts selfishly. In particular, efficient fully distributed algorithms for effective group formation are needed. Such algorithms should use only a few communication rounds among the agents,

place a very modest computational burden on each agent, and ensure that a distributed consensus among the agents on who is forming a group with whom is effectively, reliably and efficiently reached.

We propose herein one such class of distributed algorithms. We describe the generic, distributed group (coalition) formation algorithm in *Section 4* and give the pseudo-code in the *Appendix*. Variations of this basic max-clique-based group formation algorithm can be designed to meet the needs of various types of agents, such as, to give some examples, the following:

- the classical cooperative DPS agents [9, 10, 18, 19];
- different kinds of self-interested, either strictly competing or competing-and-cooperating agents [14, 24] where concepts, paradigms and tools from *N-person game theory* have found many applications; and, more generally,
- various bounded-resource, imperfect-knowledge agents acting in complex environments (e.g., [24, 30]) that are typically only *partially accessible* to any agent; such autonomous agents are thus characterized by *bounded rationality* [20].

We propose herewith a generic distributed group formation algorithm based on the idea that, in peer-to-peer (in particular, *leaderless*) *MAS*, an agent (node) would prefer to form a group with those agents that it can communicate with directly, and, moreover, where every member of such a potential group can communicate with any other member directly. That is, the preferable groups (coalitions) are actually (maximal) cliques. It is well-known that finding a maximal clique in an arbitrary graph is **NP**-complete in the centralized setting [3, 5]. This implies the computational hardness that, in general, each node faces when trying to determine maximal clique(s) it belongs to. However, if the degree of a node (that is, its number of neighbors in the graph) is small, in particular, if it is $O(1)$, then finding all maximal cliques this node belongs to is computationally feasible. If one cannot guarantee that, or *a priori* does not know if, all the nodes in a given underlying *MAS* interconnection topology are of small degree, then one has to impose additional constraints in order to ensure that the agents are not attempting to solve an infeasible problem. In particular, we shall additionally require herein that the possible coalitions to be formed, be of up to a certain pre-specified maximum size.

One may ask, why would this, maximal-clique based approach be promising for the very large scale (or *massive*) multi-agent systems (*MMAS*) that may contain ensembles of anywhere from thousands to millions of agents? The underlying graph (network topology) of such *MMAS* is bound to be very large, thus, one may argue, rendering even many typically efficient (i.e., polynomial-time in the number of agents) algorithms obsolete due to their prohibitive cost, let alone allowing distributed coordination strategies that are based on the graph theoretic algorithms that are themselves, in the classical, centralized setting, **NP**-complete in general. However, there is one critical observation that saves the day of the approach that we propose herewith: even if the underlying graph of an *MMAS* is indeed very large (possibly millions of nodes), in many important applications this graph will also tend to be *very sparse*. That is, a typical, average node (agent) will tend to have only a handful, and almost certainly only $O(1)$ neighbors. Therefore, a distributed algorithm where agents act, reason and communicate strictly locally, where no *flooding* of the network is ever performed (or needed), and where each node needs to store and work with only the data pertaining to its near-by nodes, can still be designed to be sufficiently efficient.

Some examples of the engineering, socio-technical and socio-economic systems and infrastructures that can be modeled as *MMAS* and that are also characterized by the aforementioned *sparseness* of the underlying network topology, include the following:

(i) *Large-scale* ($10^3 - 10^4$) *ensembles of micro-UAVs* (or other similar autonomous unmanned vehicles) deployed, for example, in a surveillance or a search-and-rescue mission over a sizable geographic area. Unlike the scenarios where dozens of macro UAVs are employed, where a centralized control and/or one human operator per UAV are affordable and perhaps the most efficient and robust way of deployment, in a very large-scale system of UAVs no central control is feasible or even possible, and the run-time human intervention is either minimal or nonexistent. Such micro-UAV ensembles, therefore, need to be able to coordinate, self-organize, and self-adapt to the changing environments in a truly decentralized, dynamic and autonomous manner. For more on design and simulation challenges of such large-scale ensembles of (micro-)UAVs, see, e.g., [6, 23, 24].

(ii) *Smart sensor networks* that include anywhere from thousands to millions of tiny sensors, each of which often of

only a few millimeters in size, and of a rather limited computational and communication power. In particular, the RAM memory of such sensors is currently typically of the order of Kilobytes, the flash memory range is about $1MB$, the communication bandwidth is $10^2 - 10^3$ Kilobits/second, and a typical battery life span is anywhere from a few hours up to a week. Such smart sensors usually communicate via essentially *local broadcasts* with very limited ranges. The main “communication mode” of the agents in our algorithm in *Section 4* will be precisely the local broadcasts. Also, due to small memory capacities and low power consumption requirements, smart sensors, in order to be effective, have to simultaneously minimize both the amount of local processing, and how much (and how often) they communicate messages to other sensors. Sensor networks, although on a much smaller scale, have already been used as an example to which *dynamic distributed constraint satisfaction/optimization* formalisms can be fruitfully applied [10].

(iii) Various *social networks*, and, in particular, various variants of ‘*small-world*’ *networks* where, in addition to strictly local connectivity in the communication network topology, a relatively few long-range connections are randomly added [27, 28]. A typical node in such a network will have only a handful of neighbors it can directly communicate with, and, moreover, most or nearly all of these neighbors in the network will also tend to be the neighbors in the usual, physical proximity sense.

(iv) Various *socio-technical infrastructures*, such as, e.g., traffic and transportation systems, power grids, etc. For a simple concrete example, consider a car driver (an autonomous agent) participating in the traffic in a large city. While there may be millions of such drivers on the road at the same time, the decision-making of a driver-agent is based, for the most part¹, on a few properties of its local environment, i.e., the actions of near-by agents; likewise, his own actions will usually directly affect only a handful of other agents in the system (e.g., the driver right behind him, or the pedestrian trying to cross the street right ahead of him). Thus, when modeling and simulating such an infrastructure with an appropriate large-scale network and a *MMAS*, the underlying network, while of a very large size, will be in general fairly sparse, and a typical node adjacent to (meaning, in this context, capable of directly affecting and/or perceiving) only a small number of other nodes. An ambitious project on realistic, large-scale modeling and simulation of infrastructures such as city traffic systems, called *TRANSIMS*, is described at [26] and in the documents found therein.

We now return to the dynamic, distributed group or coalition formation in massively multi-agent systems, and the mathematical formalisms and algorithmic solutions proposed for this challenging problem. A variety of coalition formation mechanisms have been proposed in the *MAS* literature both in the context of DPS agents that are all sharing the same goal (as, e.g., in [19]) and in the context of self-interested agents where each agent has its own individual agenda (as, e.g., in [18, 33]). In particular, the problem of distributed task or resource allocation, and how is this task allocation coupled to what coalition structures are (most) desirable in a given scenario [9, 19], are also of central importance in our own work on a concrete *MAS* application with a particular type of robotic agents, namely, unmanned aerial vehicles (UAVs), that are residing and acting in bounded resource multi-task environments [6, 23, 24].

Another body of *MAS* literature highly relevant to our central problem herein, namely distributed coalition formation and task and/or resource allocation, casts the distributed resource allocation problems into the distributed constraint satisfaction and/or optimization (DCS/DCO) terms [9, 10, 11].

Of the particular relevance to our work herein and other possible extensions of the original maximal clique based group formation algorithm presented in [25] are references [19] and [10]. While Modi et al. in [10] offer the most complete formalization of various distributed resource and/or task allocation problems and general mappings to appropriate types or subclasses of (dynamic) distributed constraint problems, two characteristics of their approach make it unsuitable for a direct application to our modeling framework of massively multi-agent systems in general (see, e.g., [24]), and the application domains we had in mind when devising the algorithm presented herein, in particular.

One, the agents in [10] are strictly cooperative, share the same goals, and have no notion of individual utilities or preferences. While we have studied cooperative *MAS* in [24] and elsewhere, as well, one of our main assumptions is that,

¹ There are exceptions of course; for instance, when an imminently approaching tornado is announced on the radio.

due to a large scale of the system and a high dynamism and unpredictability of the changes in the environment and the goals, no shared or global knowledge of the environment or the goals is maintained, and, in particular, each agent has its own individual preferences over the possible (local) states of the world. The collaboration is then achieved through “encoding” incentives into the individual agents’ *individual behavior functions* [23], and thus using the *incentive engineering* approach [2] to enable the agents to cooperatively coordinate even though each agent is, strictly speaking, *self-interested*.

Two, we address the issue of which agents will select which tasks to serve [23, 24], or, as in this paper, which groups of agents will form in order to serve some particular set of tasks. To that end, what is critical is an agent’s or agent coalition’s capabilities for serving the desired task(s). Each agent possesses a tuple of capabilities or available resources; likewise, each task has a tuple of resource requirements. An agent coalition can serve a particular task if and only if each component (i.e., individual capability or resource) of its joint tuple of capabilities is greater than or equal to the corresponding requirement vector entry of that task. An agent can only belong to one coalition at a time, and work with its fellow coalition members on one task at a time. Thus, while the agent operations in [10] are mutually exclusive with respect to time (an agent can only execute a single operation at any time), our agent capabilities are mutually exclusive with respect to space or, equivalently, task allocation. Clearly, a complete model of resource and task allocation in *MMAS* should incorporate both aspects. Since we are presently only interested in coalition formation for the purpose of coalition-to-task mapping, but we are not concerned herein with *how* are then these agent coalitions exactly going to perform the tasks they have been mapped to, our framework as described in [24] and outlined herein suffices for the current purposes.

The importance of DCS in *MAS* in general is discussed, e.g., in [32]. However, further discussion of DCS based approaches to distributed resource or task allocation and coalition formation is beyond the scope of this paper.

3 Problem Statement and Main Assumptions

The main purpose of this work is a generic, fully distributed, scalable and efficient algorithm for ensembles of autonomous agents to use as a subroutine - that is, as a part of their coordination strategy - with a purpose of efficiently forming temporary groups or coalitions.

The proposed algorithm is a graph algorithm. The underlying undirected graph² captures the communication (ad hoc) network topology among the agents, as follows. Each agent is a node in the graph. As for the edges, the necessary requirement for an edge between two nodes to exist is that the two nodes be able to directly communicate with one another at the time our distributed group formation subroutine is called. That is, an unordered pair of nodes $\{A, B\}$ is an edge of the underlying graph if and only if *A* can communicate messages to *B*, or *B* can communicate messages to *A*, or both.³.

The basic idea is to efficiently partition this graph into (preferably, *maximal cliques*) of nodes. These maximal cliques would usually also need to satisfy some additional criteria in order to form temporary coalitions of desired quality. These coalitions are then maintained until they are no longer useful or meaningful. For instance, the coalitions should be transformed (or else simply dissolved) when the interconnection topology of the underlying graph considerably changes, either due to the agents’ mobility, or because many old links have died out and perhaps many new, different links have formed,

² For simplicity, we assume the graph is undirected, even though the communication from one node to another is clearly directional, and the communication links need not be symmetric. However, since the nodes eventually need to reach a mutual consensus, which requires that *all* members of a coalition agree to the same coalition and *notify* all other coalition members of the agreement, the assumption about the edge (non-)directionality is inconsequential, in a sense that only those coalitions that indeed are cliques in the corresponding *directed graph* will ever be agreed upon by the agents.

³ We point out that this definition of the graph edges can be made tighter by imposing additional requirements, such as, e.g., that the two agents (that is, graph nodes), if they are to be connected by an edge, also need to be compatible, for instance, in terms of their capabilities, that they each provide some resource(s) that the other agent needs, and/or the like.

and the like. Another possible reason to abandon the existing coalition structure is when the agents determine that the coalitions have accomplished the set of tasks that these coalitions were formed to address. Thus, in an actual MAS application, the proposed group formation algorithm may need to be invoked a number of times as a coordination subroutine.

We assume that each agent has a locally accurate (see, e.g., discussion in [24]) picture of (i) who are its neighboring agents, and (ii) what are the near-by tasks and, in particular, what are these tasks' resource requirements. Each agent is equipped with a tuple of its internal resources, or *capabilities* [19]. Each task requires certain amount of each of the individual resources in this tuple in order to be serviced. A single agent, or a coalition of two or more agents, can serve a particular task if and only if their joint capabilities suffice with respect to the task's resource consumption requirements. That is, the sum of each component of the capability vector taken over all the agents in the coalition has to be greater than, or equal to, the corresponding component of the task's resource consumption vector.

Our distributed max-clique based group formation algorithm is sketched in the next section. For this algorithm to be applicable, the following basic assumptions need to hold:

- Agents communicate with one another by exchanging messages either via local broadcasts, or in a peer-to-peer fashion.

- Communication bandwidth availability is assumed not to be an issue.

- Each agent has a sufficient local memory for storing all the information received from other agents.

- Communication is reliable during the group formation, in the following sense: if an agent, A , sends a message to another agent B (either via a local broadcast where it is assumed that B is within the *communication range* of A , or in a direct, peer-to-peer manner), either agent B gets *exactly* the same message that A has sent, or else the communication link has completely failed and so B does not receive anything from A at all. In particular, we assume no *scrambled* or otherwise modified messages are ever received by any receiving agent. Also, once the groups are formed, the above assumption on communication reliability need no longer hold⁴.

- Each agent has (or else can efficiently obtain) a reliable knowledge of which other agents are within its communication range.

- Each agent, i.e., each network node, has a unique global identifier, 'UID', and the agent knows its UID.

- There is a total ordering, \prec , on the set of UIDs, and each agent knows this ordering \prec .

- Each agent uses *time-outs* in order to place an upper bound on for how long it may be waiting to hear from any other agent. If agent A has sent a message (e.g., a coalition proposal - see *Section 4*) to agent B , and the latter is not responding at all, there are three possibilities: (i) agent B has failed; (ii) communication link from B to A has failed; or (iii) while B is in A 's communication range, *vice versa* actually does not hold (but A may not know it).

- The *veracity* assumption holds, i.e., an agent can trust the information about tasks, capabilities, etc. it receives from other agents.

To summarize, when agent A sends a message to B , then B either receives exactly what A had sent, or nothing at all - and, moreover, if B has received the message, B knows that A is telling the truth about its neighbors, capabilities, commitments and preferences in it (see *Appendix*).

On the other hand, an agent need not *a priori* know the UIDs of any of the other agents, or, indeed, how many other agents are present in the system at any time.

In addition to its globally unique identifier UID, which we assume is a positive integer, and the vector of capabilities, each agent has two local flags that it uses in communication with other agents. One of the flags is the binary "decision flag", which indicates whether or not this agent has already joined some group (coalition). Namely, $decision_flag \in \{0, 1\}$, and the value of this flag is 0 as long as the agent still has not irrevocably committed to what coalition it is joining. The second flag is the "choice flag", which is used to indicate to other agents, how "happy" the agent is with its current tentative choice

⁴ As this requirement is still restrictive, and considerably limits the robustness of our algorithm, we will try to relax this assumption in our future work, and enable the agents to effectively form groups even in the presence of some limited amount of communication noise during the group formation process.

or proposal of the group to be formed. That is, the choice flag indicates the level of an agent's urgency that its proposal for a particular coalition to be formed be accepted by the neighbors to whom this proposal is being sent. For more details, we refer the reader to *Appendix* and reference [25].

4 An Outline of Max-Clique-Based Distributed Group Formation for Task Allocation

Now that the assumptions have been stated and the notation has been introduced, we outline our distributed maximal clique based coalition formation algorithm. We describe in some detail how the algorithm works below; the pseudo-code is given in *Appendix*.

The proposed distributed group or coalition formation algorithm is based on two main ideas. One idea, familiar from the literature (see, e.g., [19] and references therein), is to formulate a distributed task and/or resource allocation problem as a (*distributed*) set covering problem, $(D)SC$, in those scenarios where group overlaps are allowed, or a (*distributed*) set partitioning problem, $(D)SP$, when no group overlaps are allowed. Two (or more) groups overlap if there exists an element that belongs to both (all) of them. It is well-known that decision versions of the classical, centralized versions of the SC and SP problems are **NP**-complete [5]. Hence, we need efficient (*distributed*) heuristics so that the agents can effectively deploy DSC - or DSP -based strategies for coalition formation. Fortunately, some such efficient heuristics are already readily available [19].

The second main idea is to ensure that the formed groups of agents meet the robustness and fault tolerance criteria, which are particularly important in applications where there is a high probability of node and/or communication link failures. Indeed, we were primarily motivated by such applications when designing the algorithm proposed herewith (see [23, 24]). The most robust groups of agents of a particular size are those that correspond to *cliques* in the underlying interconnection topology of the agents' communication network. Moreover, the larger such a clique, the more robust the group of agents in the clique with respect to the node and/or link failures. Hence, appropriate maximal cliques need to be formed in a distributed fashion. However, the *Maximal Clique* problem is also well-known to be **NP**-hard [3, 5]. This hardness stems from the fact that an agent, in the general case (arbitrary underlying graph), may need to test for "cliqueness" exponentially many candidate subsets that it belongs to. However, in graphs where the maximum degree of each node is bounded by $c \log N$, where c is a constant and N is the total number of nodes in the graph, the number of subsets that each node belongs to, and therefore the number of candidate cliques, is $O(N^c)$, i.e., *polynomial in the total number of nodes*, N . In particular, in sufficiently sparse graphs, where the node degrees are bounded by some (small) constant, $K = O(1)$, the size of any maximal clique cannot exceed K . In such situations, since 2^K is presumably still sufficiently small, finding maximal cliques becomes both theoretically feasible (i.e., solvable in the time polynomial in the number of agents) and practically computable in the online, real-time scenarios that are of main interest in *MMAS* applications⁵.

We approach distributed coalition or group formation for task allocation as follows. The "candidate coalitions" are going to be required (whenever possible) to be cliques of modest sizes. That is, the system designer, based on the application at hand and the available system resources (local computational capabilities of each agent, bandwidth of agent-to-agent communication links, etc.), *a priori* chooses a threshold, K , such that only coalitions of sizes up to K are considered. Agents themselves subsequently form groups in a fully distributed and online manner, as follows. Each agent (i) first learns of who are its neighbors, then (ii) determines appropriate candidate coalitions, that the agent hopes are (preferably maximal, but certainly of size bounded by K) cliques that it belongs to, then (iii) evaluates the utility value of each such candidate coalition, measured in terms of the joint resources of all the potential coalition members, then (iv) chooses the most desirable (highest utility value to the agent) candidate coalition, and, finally, (v) sends this choice to all its neighbors.

⁵ We remark that there are also other important and frequently encountered in practice special cases, in terms of the structural properties of the underlying graphs, where the *Maximal Clique* problem turns out to be computationally feasible. In particular, the sharp uniform upper bound on all node degrees is sufficient, but not necessary, for the computational feasibility of the *Max Clique* problem.

This basic procedure is then repeated (see the *WHILE* loop in the pseudo-code in *Appendix*), together with all agents updating their knowledge of (a) what are the preferred coalitions of their neighbors, and (b) what coalitions have already been formed.

We remark that any *candidate coalition*, that is, a subset of the set of all neighbors of an agent, such that the agent currently considers this subset to be a possible choice of the coalition this agent would like to form, need not be a clique, let alone a maximal clique. Indeed, based on its strictly local knowledge (the basic information it has received, and keeps receiving, from its nearest neighbors), the agent in general does not know which of its candidate coalitions are cliques, if any. However, only those candidate coalitions that indeed *are cliques* will ever be agreed upon by the participating agents, and therefore possibly become the *actual* (as opposed to merely *potential*) coalitions. This observation justifies the name of our algorithm. Moreover, in case of the candidate coalitions of fewer than K elements (see *Appendix*) that actually end up being agreed upon by the agents, and therefore becoming the *actual coalitions*, it can be proved that these groups of nodes actually do form *maximal cliques* in a sense that these cliques can be possibly made larger in only two ways: by adding the nodes so that the threshold K is exceeded, and/or by taking away the nodes that already belong to another, equally good or better, coalition.

The algorithm proceeds in five major stages. The only assumption about *synchrony* among different nodes is that no node begins stage $n + 1$ before all the nodes have completed stage n (for $n \in \{1, 2, 3, 4, 5\}$)⁶. Within each stage, however, every node (agent) does its local computations, as well as communication (local broadcasts) independently, i.e., asynchronously and in parallel with respect to other agents. We do assume that no node failures take place during the group formation; communication links, on the other hand, are allowed to fail - but the tacit assumption is that there won't be too many such link failures, so that non-trivial clique-like groups can be formed.

The five stages of the algorithm, and a brief description of each stage, follow:

Stage 1:

Set $counter \leftarrow 1$.

Each node (in parallel) broadcasts a tuple to all its immediate neighbors. The entries in this tuple are (i) the node's UID, (ii) the node's list of (immediate) neighbors, $L(i)$, (iii) the value of the choice flag, and (iv) the value of the decision flag.

WHILE (not every agent has joined a group yet) DO

Stage 2:

Each node (in parallel) computes the overlaps of its neighborhood list with the neighborhood lists that it has received from its neighbors, $C(i, j) \leftarrow L(i) \cap L(j)$. Repetitions (if any) among this neighborhood list intersections are deleted. The remaining intersections are ordered with respect to the list size (the ties, if any, are broken arbitrarily), and a new (ordered) collection of these intersection lists (heretofore referred to simply as 'lists') is then formed.

If $counter > 1$ then:

Each node looks for information from its neighbors, whether they have joined a group "for good" during the previous round. Those neighbors that have (i.e., whose *decision-flag* = 1), are deleted from the neighborhood list $L(i)$; the intersection lists $C(i, j)$ are also updated accordingly, and those $C(i, k)$ for which k is deleted from the neighborhood list $L(i)$ are also deleted.

Stage 3:

Each remaining node (in parallel) picks one of the most preferable lists $C(i, j)$; let $C(i) \leftarrow chosen [C(i, j)]$. If the group or coalition size is the main criterion, then this means, that one of the lists of maximal length is chosen. If the combined *capabilities* of each tentative coalition for servicing various tasks is the main criterion, then each agent evaluates or estimates the *coalition value* with respect to its (possibly imperfect, and generally local) knowledge of the existing tasks and their demands in terms of resources or capabilities. To evaluate these coalition values of what are as of yet only *tentative*

⁶ We do hope to relax this assumption in future refinements of the current version of the algorithm.

coalitions, the agent needs to obtain information about other, near-by agents' capabilities. The agent then orders possible future coalitions based on these estimated coalition values, and picks as its current coalition proposal one of the possible coalitions with the highest coalition value. Since the assumption is that the capability vector of each agent has all entries nonnegative, this *monotonicity property* ensures that no proper subset of a candidate max clique coalition is ever chosen - except in the cases when the clique size exceeds the pre-specified threshold, K .

Irrespective of the exact criteria for the coalition quality, once the agent has partially ordered all candidate coalitions whose sizes do not exceed the upper bound K , it picks the best (or, in case of a tie, one of the best) coalition(s) with respect to those criteria. Then, the value of the choice flag is set, based on whether the agent has other choices of candidate coalitions that are as preferable as the current choice, and, if not, whether there are any other still available (nontrivial) choices at all.

Stage 4:

Each node (in parallel) sends its tuple with its UID, the tentatively chosen list $C(i)$, the value of the choice flag, and the value of the decision flag, to all its neighbors.

Stage 5:

Each node i (in parallel) compares its chosen list $C(i)$ with lists $C(j)$ received from its neighbors. If a clique that includes the node i exists, and all members of this clique have selected it at this stage as their current group or coalition of choice (that is, if $C(i) = C(j)$ for all $j \in C(i)$), this will be efficiently recognized by the nodes forming this clique. The decision flag of each node $j \in C(i)$ is set to 1, a group is formed, and this information is broadcast by each node in the newly formed group to all of its neighbors. Else, if no such agreement is reached, then agent i , based on its UID and priority, and its current value of the *choice flag*, either does nothing, or else changes its mind about its current group of choice, $C(i)$. The latter scenario is possible only if $\text{choice} > 0$, meaning that there are other choices of *potential* coalitions $C(i)$ that have not been tried out yet that are still available to agent i .

```
counter ← counter + 1;
END DO  [* end of WHILE loop *]
```

If $round > 1$ then, at Stage 2, each node looks for the information from its neighbors to find out if any of them have joined a group in the previous round. For those nodes that have (i.e., whose decision flag $dec = 1$), each node neighboring any such already committed node deletes this committed node from its neighborhood list $L(i)$, updates all $C(i, j)$ that remain, and selects its choice of $C(i)$ based on the updated collection of group choices $\{C(i, j) : j \in L(i)\}$. That is, now all those nodes that have already made their commitments and formed groups are not "in the game" any more, and are therefore deleted from all remaining agents' neighborhood lists as well as the tentative choices of coalitions. (Of course, the only coalition a committed agent is *not* deleted from at this stage is *the coalition* that this agent has just joined).

It can be readily shown that, once all agents exit the *WHILE* loop (that is, a repeated execution of the *Stages 2-5*), each thereby formed group is, indeed, a clique. Moreover, those agent coalitions whose sizes do not exceed the pre-specified threshold, K , are also maximal in a sense that, given such a coalition C , no agent(s) outside of this coalition can be added to it, so that (i) each of the new agents is already adjacent in the communication topology to all the "old" coalition members of C , (ii) if more than one new agent is added, then all the added agents are also pairwise neighbors to each other, (iii) the newly added agent(s) did not already belong to a coalition (or coalitions) at least as good as C , and (iv) the new size of the augmented coalition C is still at most K .

However, it is easy to construct examples of underlying graphs and particular "legal" runs of the algorithm (cf. in terms of the "tie-breaking" when an agent has two or more equally preferred choices to choose from) such that, once every node joins a coalition and the algorithm terminates, several agents end up in trivial coalitions, such as, e.g., groups of size 1 or 2. It is therefore reasonable, in most applications, to introduce an (optional) *Stage 6* of the algorithm, where these small - and therefore potentially not sufficiently robust, or useful - coalitions, whenever possible, are merged together. That is,

if some two small coalitions are adjacent to each other⁷, they can be merged together. Obviously, the connectivity of such coalitions, and therefore their tolerance to communication link failures, is in general going to be lower than that of those coalitions that are genuine cliques.

How are the non-clique coalitions to be formed, i.e., what criteria are to be used for merging together small groups into larger ones, critically depends on the nature of the underlying application and the designer's priorities when it comes to the agent coalitions' desired properties. Further discussion of this important issue, however, is beyond our current scope.

There are some more details in the algorithm that we leave out for the space constraint reasons. One important technicality is that, in order to ensure that the algorithm avoids to cycle in every possible scenario, once an agent changes its mind about the preferred coalition $C(i)$, it is not allowed through the remaining rounds of the algorithm to go back to its old choice(s). Once no other choices are left, this particular agent sticks to its current (and the only remaining) choice, and waits for other agents to converge to their choices. It can be shown that this ensures ultimate convergence to a coalition structure that all agents agree to. That is, under the assumptions stated in the previous section, the agents will reach consensus on the coalition structure after a finite number of rounds inside the WHILE loop (see also *Appendix*). Moreover, if the maximum size of any $L(i)$ is a (small) constant, then the convergence is fast.

5 Analysis and Discussion

We have outlined a fully distributed algorithm for group or coalition formation based on maximal cliques, combined with the set partition based distributed task allocation. This algorithm will be feasible when the underlying graph is relatively sparse, and, in particular, when the sizes of all maximal cliques are bounded by $c \cdot \log N$ for some small (i.e., close to 1) constant c and the number of agents N . For N very large, the proposed algorithm will be highly efficient if the maximal node degree in the underlying graph is bounded by a constant $K = O(1)$ of a modest size. When this is not the case (or when it cannot be guaranteed to always hold), appropriate restrictions can be imposed "from the outside" to ensure that the algorithm (i) converges, and (ii) is of a feasible computational complexity. Also, the coalition value for each sufficiently small subset of the set of all agents has to be efficiently computable by each agent involved.

Once the groups are formed, these groups will be tight (as everyone in the group can communicate with everyone else), and, in nontrivial cases, therefore as robust as possible for a given number of group members with respect to either node or link failures. This is a highly desirable property involving coalitions or teams of agents operating in environments where both the agent failures and the agent-to-agent communication link failures can be expected. One example of such *MAS* application domain, and in particular coordination strategies in this domain, are studied in [6, 7, 23, 24].

The proposed algorithm can be used as a subroutine in many multi-agent system scenarios where, at various points in time, the system needs to reconfigure itself, and the agents need to form new coalitions, or transform the existing ones, in a fully distributed manner, where each agent would join an appropriate (new) coalition because the agent finds this to be in its individual best interest, and where it is important for agents to agree *efficiently* on what coalitions are to be formed, rather than spending precious resources on complex negotiation protocols.

It is important, however, to point out that, in case of the self-interested agents, some form of negotiation among the agents is typically unavoidable, irrespective of the particular mechanism employed for the purposes of generating a desired coalition structure. In particular, self-interested agents that would use a variant of our algorithm presented in the previous section, would still need to negotiate, at the very least, on how the spoils to be divided up among them. According to T. Sandholm in *Chapter 5* of [29], a complete coalition formation process is made of three stages: (i) coalition structure generation, (ii) optimization within each coalition, and (iii) payoff (or utility) distribution; see also [17]. In case of the DPS agents, stages (i) and (ii) generally suffice, but when the agents have their individual preferences over the states of the world in general, and the desirability of different coalitions and tasks in particular, addressing (iii) is necessary for a coalition formation strategy to be complete and effective.

⁷ That is, if there exist node x in the first coalition and node y in the second such that x and y are adjacent in the underlying graph.

We also notice that, in our approach, the stages (i) and (ii) are not separated from each other (so that forming coalitions causally and temporally precedes the agents within each coalition then attempting to solve an appropriate optimization problem), but, instead, (i) and (ii) are *intertwined*. That is, the desired coalition structure emerges as a result of the agents solving simultaneously a coordination and a distributed constraint optimization problem. We briefly outline our view on why, in many MAS applications, stages (i) and (ii) in Sandholm’s classification scheme need not be separated as in [17]. Since in most MAS situations coordination (herein, reduced to forming groups or coalitions) is not a goal in itself, but, rather, a capability needed for, or an effective approach to, solving an underlying optimization problem, such as (distributed) resource or task allocation, the good ways to coordinate are those that enable the agents to effectively solve the underlying optimization problem. Thus the “goodness” of particular coalitions and the overall coalition structure (stage (i)), often cannot be separated from “optimization within each coalition” (stage (ii)): to optimize well *precisely means* to form good coalitions with respect to the properties of the agents’ tasks and resources.

An important game-theoretic consideration in the context of coalition formation in *N-person games* [13] is that of the nature of the environment, and, in particular, whether the environment is *super-additive*, *sub-additive*, or neither. The nature of the environment and its implications in the MAS context were formalized and discussed, e.g., in [17, 18]. Initially, in designing the maximal clique group formation algorithm [25], we have tacitly assumed *locally super-additive environments*, that is, super-additivity subject to the constraints stemming from the communication network topology that restricts which agents can communicate to each other *directly* (as opposed to via multiple hops). Local or constrained super-additivity still holds when the coalition quality is measured with respect to the *joint capabilities* of the coalition members, as long as either a “single shot” coalition-to-task mapping is performed, or, alternatively, if each agent’s resources or capabilities are renewable after each task (or round of tasks) is completed. Hence, in this context, insofar as the coalition-to-task mapping is concerned, all that matters is that the total resources of agents in the coalition exceed the resource requirements of the desired task - but by *how much* is not considered relevant. Once the agent resources are depleteable, and an agent is expected to participate in completing several tasks (with each agent or coalition still restricted to being able to work on only one task at a time), on the other hand, these agents would be interested in long-term *planning*, in computing the *marginal utility* of servicing each task [16], etc. These considerations, however, are beyond our current scope. For our purposes herein, without further ado, the environments are assumed *locally super-additive* in the sense outlined above. Once the issue of how is the payoff for servicing the tasks to be distributed among the respective coalition members is brought to the picture (stage (iii) in Sandholm’s classification), even the local (or constrained) super-additivity assumption, in general, becomes hard to justify. However, as already mentioned, in the present work we intentionally avoid addressing the issue of the payment disbursements altogether.

Last but not least, we emphasize that our algorithm as a coordination subroutine in MAS can be expected to be useful only when the time scale of significant changes in the inter-agent communication topology is much coarser than the time scale for the coalitions of agents, first, to form according to the algorithm, and, second, once formed, to accomplish something useful in terms of the agents’ ultimate goals (see, e.g., [24, 25]).

6 Summary

We have proposed herewith a generic algorithm for distributed group formation based on (maximal) cliques of modest sizes. We find this algorithm, or its appropriately fine-tuned variants, to be a potentially very useful subroutine in many multi-agent system applications, where the interconnection topology of the agents often changes so that the system needs to dynamically reconfigure itself *repeatedly*, yet where these topology changes are at a time scale that allows agents to (i) form their coalitions, and (ii) do something useful while participating in such coalitions, before the underlying communication topology of the system changes so much as to render the formed coalitions either obsolete or ineffective.

As for the future work, we plan a detailed comparative analysis of the approach presented herein on one, and the well-known coalition formation approaches known from the MAS literature, on the other hand. In particular, we would like to

compare and contrast the purely P2P, genuinely “democratic” approaches to multi-agent coordination, where *all agents are made equal* (except possibly for the different capability vectors), with the asymmetric, less democratic and more leader-based coordination approaches (such as, e.g., various automated dynamic auctions). Intuitively, the genuinely leaderless mechanisms for coalition formation, such as our maximal clique based approach, are less prone to “bottlenecks” and single points of failure than the coordination strategies where certain agents are given (even if only temporarily) special roles or “leader” status. However, this intuition needs to be both further theoretically investigated and experimentally tested and validated via appropriate comparative simulations and performance measurements.

One possible “testing ground” for determining the practical usefulness of our approach to multi-agent coordination, and its comparative (dis)advantages with respect to other approaches known from the literature, is the scalable simulation of bounded-resource autonomous unmanned aerial vehicles (UAVs) on a complex multi-task mission, developed at Open Systems Laboratory (OSL); see <http://osl.cs.uiuc.edu/> for more details. Our short-term plans, therefore, include implementation and testing of appropriately fine-tuned variants of the algorithm presented herein in the context of the OSL’s large-scale UAV simulation.

Acknowledgment: Many thanks to Myeong-wuk Jang, Nirman Kumar and Reza Ziae (all of Open Systems Laboratory, UIUC) for many useful discussions. This work was supported in part by the *DARPA IPTO TASK Program* under the contract *F30602-00-2-0586*. The first author would also like to acknowledge and express his gratitude for the travel grant from the MMAS’04 conference organizers.

References

1. N. M. Avouris, L. Gasser (eds.), ‘Distributed Artificial Intelligence: Theory and Praxis”, Euro Courses Comp. & Info. Sci. vol. 5, Kluwer Academic Publ., 1992
2. D. H. Cansever, ‘Incentive Control Strategies For Decision Problems With Parametric Uncertainties”, Ph.D. thesis, Univ. of Illinois Urbana-Champaign, 1985
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, ‘Introduction to Algorithms”, MIT Press, 1990
4. S. Franklin, A. Graesser, ‘Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents”, Proc. 3rd Int’l Workshop on Agent Theories, Architectures & Languages, Springer-Verlag, 1996
5. M. R. Garey, D. S. Johnson, ‘Computers and Intractability: a Guide to the Theory of NP-completeness”, W. H. Freedman & Co., New York, 1979
6. M. Jang, S. Reddy, P. Totic, L. Chen, G. Agha, ‘An Actor-based Simulation for Studying UAV Coordination”, Proc. 15th Euro. Symp. Simul. (ESS 2003), Delft, The Netherlands, 2003
7. M. Jang, G. Agha, ‘On Efficient Communication and Service Agent Discovery in Multi-agent Systems,” 3rd Int’l Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS ’04), pp. 27-33, May 24-25, Edinburgh, Scotland, 2004
8. N. Lynch, ‘Distributed Algorithms”, Morgan Kaufmann Publ., Wonderland, 1996
9. P. J. Modi, H. Jung, W. Shen, M. Tambe, S. Kulkarni, ‘A dynamic distributed constraint satisfaction approach to resource allocation”, in Proc. 7th Int’l Conf. on Principles & Practice of Constraint Programming, 2001
10. P. J. Modi, H. Jung, W. Shen, ‘Distributed Resource Allocation: Formalization, Complexity Results and Mappings to Distributed CSPs”, technical report (extended version of [9]), November 2002
11. P. J. Modi, W. Shen, M. Tambe, M. Yokoo, ‘An asynchronous complete method for distributed constraint optimization”, Proc. 2nd AAMAS-03, Melbourne, Australia, 2003
12. J. von Neumann, O. Morgenstern, ‘Theory of Games and Economic Behavior”, Princeton Univ. Press, 1944
13. A. Rapoport, ‘N-Person Game Theory”, The Univ. of Michigan Press, 1970
14. J. Rosenschein, G. Zlotkin, ‘Rules of Encounter: Designing Conventions for Automated Negotiations among Computers”, The MIT Press, Cambridge, Massachusetts, 1994
15. S. Russell, P. Norvig, ‘Artificial Intelligence: A Modern Approach”, 2nd ed., Prentice Hall Series in AI, 2003

16. T. Sandholm and V. Lesser, ‘Issues in automated negotiation and electronic commerce: Extending the contract net framework”, in 1st Int’l Conf. on Multiagent Systems, pp. 328-335, San Francisco, 1995.
17. T. Sandholm, V. Lesser, “Coalitions among Computationally Bounded Agents”, *Artificial Intelligence*, spec. issue on ‘Principles of MAS”, 1997
18. O. Shehory, S. Kraus, ‘Coalition formation among autonomous agents: Strategies and complexity”, Proc. MAAMAW’93, Neuchatel, Switzerland, 1993
19. O. Shehory, S. Kraus, “Task allocation via coalition formation among autonomous agents”, Proc. 14th IJCAI-95, Montreal, August 1995
20. H. A. Simon, ‘Models of Man”, J. Wiley & Sons, New York, 1957
21. R. G. Smith, “The contract net protocol: high-level communication and control in a distributed problem solver”, IEEE Trans. on Computers, 29 (12), 1980
22. G. Tel, ‘Introduction To Distributed Algorithms”, 2nd ed., Cambridge Univ. Press, 2000
23. P. Tolic, M. Jang, S. Reddy, J. Chia, L. Chen, G. Agha, ‘Modeling a System of UAVs on a Mission”, Proc. SCI 2003 (invited session), Orlando, Florida, 2003
24. P. Tolic, G. Agha, ‘Modeling Agents’ Autonomous Decision Making in Multiagent, Multitask Environments”, Proc. 1st Euro. Workshop on MAS (EUMAS 2003), Oxford, England, 2003
25. P. Tolic, G. Agha, ‘Maximal Clique Based Distributed Group Formation Algorithm for Autonomous Agent Coalitions”, Proc. Workshop on Coalitions & Teams, AAMAS ’04, New York City, New York, July 19-23, 2004
26. For more on the *TRANSIMS* project at the Los Alamos National Laboratory, go to <http://www-transims.tsasa.lanl.gov/> (The ‘Documents’ link includes a number of papers and technical reports for the period 1995 - 2001)
27. D. J. Watts, ‘Small Worlds: The Dynamics of Networks Between Order and Randomness”, Princeton Univ. Press, Princeton, N. Jersey, 1999
28. D. J. Watts, S. H. Strogatz, ‘Collective dynamics of ‘small-world’ networks”, *Nature* 393, 1998
29. G. Weiss (ed.), ‘Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence”, The MIT Press, Cambridge, Massachusetts, 1999
30. M. Wooldridge, N. Jennings, ‘Intelligent Agents: Theory and Practice”, Knowledge Engin. Rev., 1995
31. M. Yokoo, K. Hirayama, ‘Algorithms for Distributed Constraint Satisfaction: A review”, AAMAS, Vol. 3, No. 2, 2000
32. M. Yokoo, ‘Distributed Constraint Satisfaction: Foundation of Cooperation in Multi-agent Systems”, Springer, 2001
33. G. Zlotkin, J.S. Rosenschein, ‘Coalition, cryptography and stability: Mechanisms for coalition formation in task oriented domains”, Proc. AAAI’94, Seattle, Washington, 1994

7 Appendix: Pseudo-Code for Max-Clique-Based Distributed Group Formation for MAS Task Allocation

Notation:

- $i :=$ the i -th agent (node) in the system (say, $i = 1, \dots, n$)
- $V(i) :=$ the i -th node's UID
- $N(i) :=$ the list of neighbors of node i
- $L(i) :=$ the extended neighborhood list (i.e., $L(i) = N(i) \cup \{i\}$)
- $C(i, j) = L(i) \cap L(j)$
- $C(i) :=$ the group of choice of node i at the current stage (i.e., one of the $C(i, j)$'s)
- $choice(i) :=$ the choice flag of node i
- $dec(i) :=$ the decision flag of node i

Remark: For simplicity, clarity and space limitations, we focus on distributed computation of cliques, and consensus reaching on what clique-like coalitions are to be formed. We therefore assume that there is a highly efficient, readily available subroutine for each agent to evaluate (or estimate) the utility value of each potential coalition. This subroutine is called independently by each agent inside of *Stage 3* below.

Max Clique-based Distributed Coalition Formation Algorithm:

Stage 1:

```
DOALL  $i = 1..n$  (in parallel, i.e., each node  $i$  carries the steps below locally)
    send  $[V(i), L(i), choice = 3, dec = 0]$  to each of your neighbors
END DOALL
```

```
WHILE (not all agents have joined a group) DO
    [ $\star$  Stages 2-5 are repeated until consensus on the coalition structure is reached  $\star$ ]
```

Stage 2:

```
DOALL  $i = 1..n$ 
    FOR all  $j \in N(i)$  DO [ $\star$  check if  $dec(j) = 1$   $\star$ ]
        if  $dec(j) == 1$  then delete  $j$  from  $N(i), L(i)$ , and  $C(i, j')$ ,  $\forall j' \in N(i) - \{j\}$ 
    END DO [ $\star$  end of FOR loop  $\star$ ]
    FOR all  $j \in N(i)$  DO [ $\star$  FOR all remaining (undeleted) indices  $j$   $\star$ ]
        compute  $C(i, j) \leftarrow L(i) \cap L(j)$ 
    END DO [ $\star$  end of FOR loop  $\star$ ]
END DOALL
```

Stage 3:

```
DOALL  $i = 1..n$ 
    FOR all  $j \in N(i)$  DO
        compute utility value  $val[C(i, j)]$  of each candidate coalition  $C(i, j)$ 8
    END DO
    pick  $C(i, l)$  such that  $val[C(i, l)] = \max_{j \in N(i)} val[C(i, j)]$  (subject to  $|C(i, j)| \leq K$ )
     $C(i) \leftarrow C(i, l);$ 
```

⁸ We require throughout, that all groups to be considered, that is, all ‘candidate coalitions’, be of appropriately bounded size, $|C(i, j)| \leq K$.

```

if (there is more than one such choice of max. utility value) then
    set choice( $i$ )  $\leftarrow 2$ ;
else (if there are other choices  $C(i, j')$  but only of strictly smaller utility value)
    set choice  $\leftarrow 1$ ;
else (if node  $i$  has no alternatives left for a non-trivial coalition that would include  $i$ )
    set choice  $\leftarrow 0$ ;
END DOALL

```

Stage 4:

```

DOALL  $i = 1..n$ 
    send [ $V(i), C(i)$ , choice, dec = 0]
END DOALL

```

Stage 5:

```

DOALL  $i = 1..n$ 
    compare  $C(i)$  with  $C(j)$  received from one's neighbors  $j \in N(i)$ ;
    if (there exists a clique  $\{i, j_1, j_2, \dots, j_l\}$  such that  $C(i) = C(j_1) = C(j_2) = \dots = C(j_l)$ )
        then set dec  $\leftarrow 1$  (an agreement has been reached);
        broadcast group  $G = \{i, j_1, j_2, \dots, j_l\}$  and dec( $i$ ) = 1 to all neighbors  $j \in N(i)$ :
        send [ $V(i), C(i) = G$ , choice, dec = 1]
    else (based on UID  $i$  and the priority as defined by the relation  $\prec$ )
        either DO NOTHING
        or change your mind:  $C(i) \leftarrow$  new choice  $C(i, j)$ 
            (from the list of candidate groups that are still available)
    END DOALL
END DO [* end of WHILE loop *]

```

Generating Optimal Monitors for Extended Regular Expressions

Koushik Sen ^{1,3}

*Department of Computer Science,
University of Illinois at Urbana Champaign,
USA.*

Grigore Roșu ^{2,4}

*Department of Computer Science,
University of Illinois at Urbana Champaign,
USA.*

Abstract

Ordinary software engineers and programmers can easily understand regular patterns, as shown by the immense interest in and the success of scripting languages like Perl, based essentially on regular expression pattern matching. We believe that regular expressions provide an elegant and powerful specification language also for *monitoring requirements*, because an execution trace of a program is in fact a string of states. Extended regular expressions (EREs) add complementation to regular expressions, which brings additional benefits by allowing one to specify patterns that must *not* occur during an execution. Complementation gives one the power to express patterns on strings more compactly. In this paper we present a technique to generate *optimal monitors* from EREs. Our monitors are deterministic finite automata (DFA) and our novel contribution is to generate them using a modern coalgebraic technique called *coinduction*. Based on experiments with our implementation, which can be publicly tested and used over the web, we believe that our technique is more efficient than the simplistic method based on complementation of automata which can quickly lead to a highly-exponential state explosion.

¹ Supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586 and the DARPA IXO NEST Program, contract number F33615-01-C-1907) and the ONR Grant N00014-02-1-0715.

² Supported in part by the joint NSF/NASA grant CCR-0234524.

³ Email: ksen@cs.uiuc.edu

⁴ Email: grosu@cs.uiuc.edu

1 Introduction

Regular expressions can express patterns in strings in a compact way. They proved very useful in practice; many programming/scripting languages like Perl, Python, Tcl/Tk support regular expressions as core features. Because of their power to express a rich class of patterns, regular expressions, are used not only in computer science but also in various other fields, such as molecular biology [18]. All these applications boast of very efficient implementation of regular expression pattern matching and/or membership algorithms. Moreover, it has been found that compactness of regular expressions can be increased non-elementarily by adding complementation ($\neg R$) to the usual union ($R_1 + R_2$), concatenation ($R_1 \cdot R_2$), and repetition (R^*) operators of regular expressions. These are known as *extended regular expressions* (EREs) and they proved very intuitive and succinct in expressing regular patterns.

Recent trends have shown that the software analysis community is inclining towards scalable techniques for software verification. Works in [12] merged temporal logics with testing, thereby getting the benefits of both worlds. The Temporal Rover tool (TR) and its follower DB Rover [5] are already commercial. In these tools the Java code is instrumented automatically so that it can check the satisfaction of temporal logic properties at runtime. The MaC tool [17,22] has been developed to monitor safety properties in interval past time temporal logics. In [24,25], various algorithms to generate testing automata from temporal logic formulae, are described. Java PathExplorer [10] is a runtime verification environment currently under development at NASA Ames. The Java MultiPathExplorer tool [29] proposes a technique to monitor all equivalent traces that can be extracted from a given execution, thus increasing the coverage of monitoring. [7,11] present efficient algorithms for monitoring future time temporal logic formulae, while [13] gives a technique to synthesize efficient monitors from past time temporal formulae. [27] uses rewriting to perform runtime monitoring of EREs.

An interesting aspect of EREs is that they can express safety properties compactly, like those encountered in testing and monitoring. By generating automata from logical formulae, several of the works above show that the safety properties expressed by different variants of temporal logics are subclasses of regular languages. The converse is *not* true, because there are regular patterns which cannot be expressed using temporal logics, most notoriously those related to counting; e.g., the regular expression $(0 \cdot (0 + 1))^*$ saying that every other letter is 0 does not admit an equivalent temporal logic formula. Additionally, EREs tend to be often very natural and intuitive in expressing requirements. For example, let us try to capture the safety property “it should not be the case that in any trace of a traffic light we see green and then immediately red at any point”. The natural and intuitive way to express it in ERE is $\neg((\neg\emptyset) \cdot \text{green} \cdot \text{red} \cdot (\neg\emptyset))$, where \emptyset is the empty ERE (no words), so $\neg\emptyset$ means “anything”.

Previous approaches to ERE membership testing [14,23,31,21,16] have focussed on developing techniques that are polynomial in both the size of the word and the size of the formulae. The best known result in these approaches is described in [21] where they can check if a word satisfies an ERE in time $O(m \cdot n^2)$ and space $O(m \cdot m + k \cdot n^2)$, where m is the size of the ERE, n is the length of the word, and k is the number of negation/intersection operators. These algorithms, unfortunately, cannot be used for the purpose of monitoring. This is because they are not incremental. They assume the entire word is available before their execution. Additionally, their running time and space requirements are quadratic in the size of the trace. This is unacceptable when one has a long trace of events and wants to monitor a small ERE, as it is typically the case. This problem is removed in [27] where traces are checked against EREs through incremental rewriting. At present, we do not know if the technique in [27] is optimal or not.

A simple, straightforward, and practical approach is to generate optimal *deterministic finite automata* (DFA) from EREs [15]. This process involves the conversion of each negative sub-component of the ERE to a non-deterministic finite automaton (NFA), determinization of the NFA into a DFA, complementation of the DFA, and then its minimization. The algorithm runs in a bottom-up fashion starting from the innermost negative ERE sub components. This method, although generates the minimal automata, is too complex and cumbersome in practice. Its space requirements can be non-elementarily larger than the initial regular ERE, because negation involves an NFA-to-DFA translation, which implies an exponential blow-up; since negations can be nested, the size of such NFAs or DFAs could be highly exponential.

Our approach is to generate the minimal DFA from an ERE using coinductive techniques. In this paper, the DFA thus generated is called the *optimal monitor* for the given ERE. A number of related approaches for simple regular expressions can be found in [30]. Our algorithm extends these approaches for simple regular expressions to EREs; further, we make the algorithm efficient by intelligent book keeping. The complexity of our algorithm seems to be hard to evaluate, because it depends on the size of the minimal DFA associated to an ERE and we are not aware of any lower bound results in this direction. However, experiments are very encouraging. Our implementation, which is available for evaluation on the internet via a CGI server reachable from <http://fsl.cs.uiuc.edu/rv/>, rarely took longer than one second to generate a DFA, and it took only 18 minutes to generate the minimal 107 state DFA for the ERE in Example 5.3 which was used to show the exponential space lower bound of ERE monitoring in [27].

In a nutshell, in our approach we use the concept of derivatives of an ERE, as described in Subsection 2.2. For a given ERE one generates all possible derivatives of the ERE for all possible sequences of events. The size of this set of derivatives depends upon the size of the initial ERE. However, several of these derivative EREs can be equivalent to each other. One can check the

equivalence of EREs using coinductive technique as described in Section 3, that generates a set of equivalent EREs, called *circularities*. In Section 4, we show how circularities can be used to construct an efficient algorithm that generates optimal DFAs from EREs. In Section 5, we describe an implementation of this algorithm and give performance analysis results. We also made available on the internet a CGI interface to this algorithm.

2 Extended Regular Expressions and Derivatives

In this section we recall extended regular expressions and their derivatives.

2.1 Extended Regular Expressions

Extended regular expressions (ERE) define languages by inductively applying union ($+$), concatenation (\cdot), Kleene Closure ($*$), intersection (\cap), and complementation (\neg). More precisely, for an alphabet E , whose elements are called *events* in this paper, an ERE over E is defined as follows, where $a \in E$:

$$R ::= \emptyset \mid \epsilon \mid a \mid R + R \mid R \cdot R \mid R^* \mid R \cap R \mid \neg R.$$

The language defined by an expression R , denoted by $\mathcal{L}(R)$, is defined inductively as

$$\begin{aligned} \mathcal{L}(\emptyset) &= \emptyset, \\ \mathcal{L}(\epsilon) &= \{\epsilon\}, \\ \mathcal{L}(A) &= \{A\}, \\ \mathcal{L}(R_1 + R_2) &= \mathcal{L}(R_1) \cup \mathcal{L}(R_2), \\ \mathcal{L}(R_1 \cdot R_2) &= \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(R_1) \text{ and } w_2 \in \mathcal{L}(R_2)\}, \\ \mathcal{L}(R^*) &= (\mathcal{L}(R))^*, \\ \mathcal{L}(R_1 \cap R_2) &= \mathcal{L}(R_1) \cap \mathcal{L}(R_2), \\ \mathcal{L}(\neg R) &= \Sigma^* \setminus \mathcal{L}(R). \end{aligned}$$

Given an ERE, as defined above using union, concatenation, Kleene Closure, intersection and complementation, one can translate it into an equivalent expression that does not have any intersection operation, by applying De Morgan's Law: $R_1 \cap R_2 = \neg(\neg R_1 + \neg R_2)$. The translation only results in a linear blowup in size. Therefore, in the rest of the paper we do not consider expressions containing intersection. More precisely, we only consider EREs of the form

$$R ::= R + R \mid R \cdot R \mid R^* \mid \neg R \mid a \mid \epsilon \mid \emptyset.$$

2.2 Derivatives

In this subsection we recall the notion of *derivative*, or “residual” (see [2,1], where several interesting properties of derivatives are also presented). It is based on the idea of “event consumption”, in the sense that an extended regular expression R and an event a produce another extended regular expression, denoted $R\{a\}$, with the property that for any trace w , $aw \in R$ if and only if $w \in R\{a\}$.

In the rest of the paper assume defined the typical operators on EREs and consider that the operator $_ + _$ is associative and commutative and that the operator $_ \cdot _$ is associative. In other words, reasoning is performed modulo the equations:

$$\begin{aligned} (R_1 + R_2) + R_3 &= R_1 + (R_2 + R_3), \\ R_1 + R_2 &= R_2 + R_1, \\ (R_1 \cdot R_2) \cdot R_3 &= R_1 \cdot (R_2 \cdot R_3). \end{aligned}$$

We next consider an operation $\{_ a\}$ which takes an ERE and an event, and give several equations which define its operational semantics recursively, on the structure of regular expressions:

$$(R_1 + R_2)\{a\} = R_1\{a\} + R_2\{a\} \quad (1)$$

$$(R_1 \cdot R_2)\{a\} = (R_1\{a\}) \cdot R_2 + \text{if } (\epsilon \in R_1) \text{ then } R_2\{a\} \text{ else } \emptyset \text{ fi} \quad (2)$$

$$(R^*)\{a\} = (R\{a\}) \cdot R^* \quad (3)$$

$$(\neg R)\{a\} = \neg(R\{a\}) \quad (4)$$

$$b\{a\} = \text{if } (b == a) \text{ then } \epsilon \text{ else } \emptyset \text{ fi} \quad (5)$$

$$\epsilon\{a\} = \emptyset \quad (6)$$

$$\emptyset\{a\} = \emptyset \quad (7)$$

The right-hand sides of these equations use operations which we describe next. “**if** ($_$) **then** $_$ **else** $_$ **fi**” takes a boolean term and two EREs as arguments and has the expected meaning defined by two equations:

$$\text{if } (\text{true}) \text{ then } R_1 \text{ else } R_2 \text{ fi} = R_1 \quad (8)$$

$$\text{if } (\text{false}) \text{ then } R_1 \text{ else } R_2 \text{ fi} = R_2 \quad (9)$$

We assume a set of equations that properly define boolean expressions and reasoning. Boolean expressions include the constants *true* and *false*, as well as the usual connectors $_ \wedge _$, $_ \vee _$, and *not*. Testing for empty trace membership (which is used by (2)) can be defined via the following equations:

$$\epsilon \in (R_1 + R_2) = (\epsilon \in R_1) \vee (\epsilon \in R_2) \quad (10)$$

$$\epsilon \in (R_1 \cdot R_2) = (\epsilon \in R_1) \wedge (\epsilon \in R_2) \quad (11)$$

$$\epsilon \in (R^*) = \text{true} \quad (12)$$

$$\epsilon \in (\neg R) = \text{not}(\epsilon \in R) \quad (13)$$

$$\epsilon \in b = \text{false} \quad (14)$$

$$\epsilon \in \epsilon = \text{true} \quad (15)$$

$$\epsilon \in \emptyset = \text{false} \quad (16)$$

The 16 equations above are natural and intuitive. [27] shows that these equations, when regarded as rewriting rules are terminating and ground Church-Rosser (modulo associativity and commutativity of $_ + _$ and modulo associativity of $_ \cdot _$), so they can be used as a functional procedure to calculate derivatives. Due to the fact that the 16 equations defining the derivatives can generate useless terms, in order to keep EREs compact we also propose defining several *simplifying equations*, including at least the following:

$$\emptyset + R = R,$$

$$\emptyset \cdot R = \emptyset,$$

$$\epsilon \cdot R = R,$$

$$R + R = R.$$

The following result (see, e.g., [27] for a proof) gives a simple procedure, based on derivatives, to test whether a word belongs to the language of an ERE:

Theorem 2.1 *For any ERE \mathcal{R} and any events a, a_1, a_2, \dots, a_n in A , the following hold:*

- 1) $a_1 a_2 \dots a_n \in \mathcal{L}(R\{a\})$ if and only if $aa_1 a_2 \dots a_n \in \mathcal{L}(R)$; and
- 2) $a_1 a_2 \dots a_n \in \mathcal{L}(R)$ if and only if $\epsilon \in R\{a_1\}\{a_2\} \dots \{a_n\}$.

3 Hidden Logic and Coinduction

We use circular coinduction, defined rigorously in the context of hidden logics and implemented in the BOBJ system [26,8,9], to test whether two EREs are equivalent, that is, if they have the same language. Since the goal of this paper is to translate an ERE into a minimal DFA, standard techniques for checking equivalence, such as translating the two expressions into DFAs and then comparing those, do not make sense in this framework. A particularly appealing aspect of circular coinduction in the framework of EREs is that it does not only show that two EREs are equivalent, but also generates a larger

set of equivalent EREs which will all be used in order to generate the target DFA.

Hidden logic is a natural extension of algebraic specification which benefits of a series of generalizations in order to capture various natural notions of behavioral equivalence found in the literature. It distinguishes *visible* sorts for data from *hidden* sorts for states, with states *behaviorally equivalent* if and only if they are indistinguishable under a formally given set of experiments. To keep the presentation simple and self contained, in this section we define an oversimplified version of hidden logic together with its associated circular coinduction proof rule, still general enough to support defining and proving EREs behaviorally equivalent.

3.1 Algebraic Preliminaries

The reader is assumed familiar with basic equational logic and algebra in this section. We recall a few notions in order to just make our notational conventions precise. An S -sorted signature Σ is a set of sorts/types S together with operational symbols on those, and a Σ -algebra A is a collection of sets $\{A_s \mid s \in S\}$ and a collection of functions appropriately defined on those sets, one for each operational symbol. Given an S -sorted signature Σ and an S -indexed set of variables Z , let $T_\Sigma(Z)$ denote the Σ -term algebra over variables in Z . If $V \subseteq S$ then $\Sigma|_V$ is a V -sorted signature consisting of all those operations in Σ with sorts entirely in V . We may let $\sigma(X)$ denote the term $\sigma(x_1, \dots, x_n)$ when the number of arguments of σ and their order and sorts are not important. If only one argument is important, then to simplify writing we place it at the beginning; for example, $\sigma(t, X)$ is a term having σ as root with only variables as arguments except one, and we do not care which one, which is t . If t is a Σ -term of sort s' over a special variable $*$ of sort s and A is a Σ -algebra, then $A_t : A_s \rightarrow A_{s'}$ is the usual interpretation of t in A .

3.2 Behavioral Equivalence, Satisfaction and Specification

Given disjoint sets V, H called *visible* and *hidden sorts*, a *hidden* (V, H) -signature, say Σ , is a many sorted $(V \cup H)$ -signature. A *hidden subsignature* of Σ is a hidden (V, H) -signature Γ with $\Gamma \subseteq \Sigma$ and $\Gamma|_V = \Sigma|_V$. The *data signature* is $\Sigma|_V$. An operation of visible result not in $\Sigma|_V$ is called an *attribute*, and a hidden sorted operation is called a *method*.

Unless otherwise stated, the rest of this section assumes fixed a hidden signature Σ with a fixed subsignature Γ . Informally, Σ -algebras are universes of possible states of a system, i.e., “black boxes,” where one is only concerned with behavior under experiments with operations in Γ , where an experiment is an observation of a system attribute after perturbation; this is formalized below.

A Γ -context for sort $s \in V \cup H$ is a term in $T_\Gamma(\{*\} : s)$ with one occurrence

of $*$. A Γ -context of visible result sort is called a Γ -*experiment*. If c is a context for sort h and $t \in T_{\Sigma,h}$ then $c[t]$ denotes the term obtained from c by substituting t for $*$; we may also write $c[*$] for the context itself.

Given a hidden Σ -algebra A with a hidden subsignature Γ , for sorts $s \in (V \cup H)$, we define Γ -behavioral equivalence of $a, a' \in A_s$ by $a \equiv_{\Sigma}^{\Gamma} a'$ iff $A_c(a) = A_c(a')$ for all Γ -experiments c ; we may write \equiv instead of \equiv_{Σ}^{Γ} when Σ and Γ can be inferred from context. We require that all operations in Σ are compatible with \equiv_{Σ}^{Γ} . Note that behavioral equivalence is the identity on visible sorts, since the trivial contexts $* : v$ are experiments for all $v \in V$. A major result in hidden logics, underlying the foundations of coinduction, is that Γ -behavioral equivalence is the largest equivalence which is identity on visible sorts and which is compatible with the operations in Γ .

Behavioral satisfaction of equations can now be naturally defined in terms of behavioral equivalence. A hidden Σ -algebra A Γ -behaviorally satisfies a Σ -equation $(\forall X) t = t'$, say e , iff for each $\theta : X \rightarrow A$, $\theta(t) \equiv_{\Sigma}^{\Gamma} \theta(t')$; in this case we write $A \models_{\Sigma}^{\Gamma} e$. If E is a set of Σ -equations we then write $A \models_{\Sigma}^{\Gamma} E$ when A Γ -behaviorally satisfies each Σ -equation in E . We may omit Σ and/or Γ from $\models_{\Sigma}^{\Gamma}$ when they are clear.

A behavioral Σ -specification is a triple (Σ, Γ, E) where Σ is a hidden signature, Γ is a hidden subsignature of Σ , and E is a set of Σ -sentences equations. Non-data Γ -operations (i.e., in $\Gamma - \Sigma|_V$) are called *behavioral*. A Σ -algebra A behaviorally satisfies a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ iff $A \models_{\Sigma}^{\Gamma} E$, in which case we write $A \models \mathcal{B}$; also $\mathcal{B} \models e$ iff $A \models \mathcal{B}$ implies $A \models_{\Sigma}^{\Gamma} e$.

EREs can be very naturally defined as a behavioral specification. The enormous benefit of doing so is that the behavioral inference, including most importantly coinduction, provide a *decision procedure* for equivalence of EREs. [8] shows how standard regular expressions (without negation) can be defined as a behavioral specification, a BOBJ implementation, and also how BOBJ with its circular coinductive rewriting algorithm can prove automatically several equivalences of regular expressions. Related interesting work can also be found in [28]. In this paper we extend that to general EREs, generate minimal observer monitors, and also give several other examples.

Example 3.1 A behavioral specification of EREs defines a set of two visible sorts $V = \{Bool, Event\}$, one hidden sort $H = \{Ere\}$, one behavioral attribute $\epsilon \in _ : Ere \rightarrow Bool$ and one behavioral method, the derivative, $_ \{-\} : Ere \times Event \rightarrow Ere$, together with all the other operations in Subsection 2.1 defining EREs, including the events in E which are defined as visible constants of sort $Event$, and all the equations in Subsection 2.2. We call it the *ERE behavioral specification* and let \mathcal{B}_{ERE} denote it.

Since the only behavioral operators are the test for ϵ membership and the derivative, it follows that the experiments have exactly the form $\epsilon \in * \{a_1\} \{a_2\} \dots \{a_n\}$, for any events a_1, a_2, \dots, a_n . In other words, an experiment consists of a series of derivations followed by an ϵ membership test, and there-

fore two regular expressions are *behavioral equivalent* if and only if they cannot be distinguished by such experiments. Notice that the above reasoning applies within *any algebra* satisfying the presented behavioral specification. The one we are interested in is, of course, the *free* one, whose set carriers contain exactly the extended regular expressions as presented in Subsection 2.1, and the operations have the obvious interpretations. We informally call it the *ERE algebra*.

Letting \equiv denote the behavioral equivalence relation generated on the ERE algebra, then Theorem 2.1 immediately yields the following important result.

Theorem 3.2 *If R_1 and R_2 are two EREs then $R_1 \equiv R_2$ if and only if $\mathcal{L}(R_1) = \mathcal{L}(R_2)$.*

This theorem allows us to prove equivalence of EREs by making use of behavioral inference in the ERE behavioral specification, from now on simply referred to by \mathcal{B} , including (especially) circular coinduction. The next section shows how circular coinduction works and how it can be used to show EREs equivalent.

3.3 Circular Coinduction as an Inference Rule

In the simplified version of hidden logics defined above, the usual equational inference rules, i.e., reflexivity, symmetry, transitivity, substitution and congruence [26] are all sound for behavioral satisfaction. However, equational reasoning can derive only a very limited amount of interesting behavioral equalities. For that reason, *circular coinduction* has been developed as a very powerful automated technique to show behavioral equivalence. We let \Vdash denote the relation being defined by the equational rules plus circular coinduction, for deduction from a specification to an equation.

Before we present circular coinduction formally, we give the reader some intuitions by duality to structural induction. The reader who is only interested in using the presented procedure or who is not familiar with structural induction, can skip this paragraph. Inductive proofs show equality of terms $t(x), t'(x)$ over a given variable x (seen as a constant) by showing $t(\sigma(x))$ equals $t'(\sigma(x))$ for all σ in a basis, while circular coinduction shows terms t, t' behaviorally equivalent by showing equivalence of $\delta(t)$ and $\delta(t')$ for all behavioral operations δ . Coinduction applies behavioral operations at the top, while structural induction applies generator/constructor operations at the bottom. Both induction and circular coinduction assume some “frozen” instances of t, t' equal when checking the inductive/coinductive step: for induction, the terms are frozen at the bottom by replacing the induction variable by a constant, so that no other terms can be placed beneath the induction variable, while for coinduction, the terms are frozen at the top, so that they cannot be used as subterms of other terms (with some important but subtle exceptions which are not needed here; see [9]).

Freezing terms at the top is elegantly handled by a simple trick. Suppose every specification has a special visible sort b , and for each (hidden or visible) sort s in the specification, a special operation $[_]: s \rightarrow b$. No equations are assumed for these operations and no user defined sentence can refer to them; they are there for technical reasons. Thus, with just the equational inference rules, for any behavioral specification \mathcal{B} and any equation $(\forall X) t = t'$, it is necessarily the case that $\mathcal{B} \Vdash (\forall X) t = t'$ iff $\mathcal{B} \Vdash (\forall X) [t] = [t']$. The rule below preserves this property. Let the sort of t, t' be hidden; then

Circular Coinduction:

$$\frac{\mathcal{B} \cup \{(\forall X) [t] = [t']\} \Vdash (\forall X, W) [\delta(t, W)] = [\delta(t', W)], \text{ for all appropriate } \delta \in \Gamma}{\mathcal{B} \Vdash (\forall X) t = t'}$$

We call the equation $(\forall X) [t] = [t']$ added to \mathcal{B} a **circularity**; it could just as well have been called a coinduction hypothesis or a co-hypothesis, but we find the first name more intuitive because from a coalgebraic point of view, coinduction is all about finding circularities.

Theorem 3.3 *The usual equational inference rules together with Circular Coinduction are sound. That means that if $\mathcal{B} \Vdash (\forall X) t = t'$ and $\text{sort}(t, t') \neq b$, or if $\mathcal{B} \Vdash (\forall X) [t] = [t']$, then $\mathcal{B} \models (\forall X) t = t'$.*

Example 3.4 Suppose that we want to show that the EREs $(a + b)^*$ and $(a^*b^*)^*$ admit the same language. By Theorem 3.2, we can instead show that $\mathcal{B}_{ERE} \models (\forall \emptyset) (a + b)^* = (a^*b^*)^*$. Notice that a and b are treated as constant events here; one can also prove the result when a and b are variables, but one would need to first make use of the theorem of hidden constants [26]. To simplify writing, we omit the empty quantifier of equations. By the Circular Coinduction rule, one generates the following three proof obligations

$$\begin{aligned} \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [\epsilon \in (a + b)^*] = [\epsilon \in (a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*\{a\}] = [(a^*b^*)^*\{a\}], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*\{b\}] = [(a^*b^*)^*\{b\}]. \end{aligned}$$

The first proof task follows immediately by using the equations in \mathcal{B} as rewriting rules, while the other two tasks reduce to

$$\begin{aligned} \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [a^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [b^*(a^*b^*)^*]. \end{aligned}$$

By applying Circular Coinduction twice, after simplifying the two obvious proof tasks stating the ϵ membership, one gets the following four proof obligations

$$\begin{aligned}\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a+b)^*]\{a\} = [a^*(a^*b^*)^*\{a\}], \\ \mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a+b)^*]\{b\} = [a^*(a^*b^*)^*\{b\}], \\ \mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a+b)^*]\{a\} = [b^*(a^*b^*)^*\{a\}], \\ \mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a+b)^*]\{b\} = [b^*(a^*b^*)^*\{b\}],\end{aligned}$$

which, after simplification translate into

$$\begin{aligned}\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a+b)^*] = [a^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a+b)^*] = [b^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a+b)^*] = [a^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a+b)^*] = [b^*(a^*b^*)^*],\end{aligned}$$

Again by applying circular coinduction we get

$$\begin{aligned}\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} &\Vdash \\ &[(a+b)^*] = [a^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} &\Vdash \\ &[(a+b)^*] = [b^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} &\Vdash \\ &[(a+b)^*] = [a^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} &\Vdash \\ &[(a+b)^*] = [b^*(a^*b^*)^*],\end{aligned}$$

which now follow all immediately. Notice that BOBJ uses the newly added (to \mathcal{B}_{ERE}) equations as rewriting rules when it applies its circular coinductive rewriting algorithm, so the proof above is done slightly differently, but entirely automatically.

Example 3.5 Suppose now that one wants to show that $\neg(a^*b) \equiv \epsilon + a^* + (a+b)^*b(a+b)(a+b)^*$. One can also do it entirely automatically by circular coinduction as above, generating the following list of circularities:

$$\begin{aligned}[\neg(a^*b)] &= [\epsilon + a^* + (a+b)^*b(a+b)(a+b)^*], \\ [\neg(\epsilon)] &= [(a+b)^*b(a+b)(a+b)^* + (a+b)(a+b)^*], \\ [\neg(\emptyset)] &= [(a+b)^*b(a+b)(a+b)^* + (a+b)^*], \\ [\neg(\emptyset)] &= [(a+b)^*b(a+b)(a+b)^* + (a+b)(a+b)^* + (a+b)^*].\end{aligned}$$

Example 3.6 One can also show by circular coinduction that concrete EREs satisfy systems of guarded equations. This is an interesting but unrelated subject, so we do not discuss it in depth here. However, we show how easily one can prove by coinduction that a^*b is the solution of the equation $R = a \cdot R + b$. This equation can be given by adding a new ERE constant r to \mathcal{B}_{ERE} , together with the equations $\epsilon \in r = \text{false}$, $r\{a\} = r$, and $r\{b\} = \epsilon$. Circular Coinduction applied on the goal $r = a^*b$ generates the proof tasks:

$$\begin{aligned}\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} &\Vdash [\epsilon \in r] = [\epsilon \in a^*b], \\ \mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} &\Vdash [r\{a\}] = [a^*b\{a\}], \\ \mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} &\Vdash [r\{b\}] = [a^*b\{b\}],\end{aligned}$$

which all follow immediately.

The following says that circular coinduction provides a decision procedure for equivalence of EREs.

Theorem 3.7 *If R_1 and R_2 are two EREs, then $\mathcal{L}(R_1) = \mathcal{L}(R_2)$ if and only if $\mathcal{B}_{ERE} \Vdash R_1 = R_2$. Moreover, since the rules in \mathcal{B}_{ERE} are ground Church-Rosser and terminating, circular coinductive rewriting [8,9], which iteratively rewrites proof tasks to their normal forms followed by a one step coinduction if needed, gives a decision procedure for ERE equivalence.*

4 Generating Minimal DFA Monitors by Coinduction

In this section we show how one can use the set of circularities generated by applying the circular coinduction rules in order to generate a minimal DFA from any ERE. This DFA can then be used as an optimal monitor for that ERE. The main idea here is to associate states in DFA to EREs obtained by deriving the initial ERE; when a new ERE is generated, it is tested for equivalence with all the other already generated EREs by using the coinductive procedure presented in the previous section. A crucial observation which significantly reduces the complexity of our procedure is that, once an equivalence is proved by circular coinductive rewriting, the entire set of circularities accumulated represent equivalent EREs. These can be used to later quickly infer the other equivalences, without having to generate the same circularities over and over again.

Since BOBJ does not (yet) provide any mechanism to return the set of circularities accumulated after proving a given behavioral equivalence, we were unable to use BOBJ to implement our optimal monitor generator. Instead, we have implemented our own version of coinductive rewriting engine for EREs, which is described below.

We are given an initial ERE R_0 over alphabet A and from that we want to generate the equivalent minimal DFA $D = (S, A, \delta, s_0, F)$, where S is the

set of states, $\delta : S \times A \rightarrow S$ is the transition function, s_0 is the initial state, and $F \subseteq S$ is the set of final states. The coinductive rewriting engine explicitly accumulates the proven circularities in a set. The set is initialized to an empty set at the beginning of the algorithm. It is updated with the accumulated circularities whenever we prove equivalence of two regular expressions in the algorithm. The algorithm maintains the set of states S in the form of non-equivalent EREs. At the beginning of the algorithm S is initialized with a single element, which is the given ERE R_0 . Next, we generate all the derivatives of the initial ERE one by one in a depth first manner. A derivative $R_x = R\{x\}$ is added to the set S , if the set does not contain any ERE equivalent to the derivative R_x . We then extend the transition function by setting $\delta(R, x) = R_x$. If an ERE R' equivalent to the derivative already exists in the set S , we extend the transition function by setting $\delta(R, x) = R'$. To check if an ERE equivalent to the derivative R_x already exists in the set S , we sequentially go through all the elements of the set S and try to prove its equivalence with R_x . In testing the equivalence we first add the set of circularities to the initial \mathcal{B} . Then we invoke the coinductive procedure. If for some ERE $R' \in S$, we are able to prove that $R' \equiv R_x$ i.e $\mathcal{B} \cup Eq_{all} \cup Eq_{new} \Vdash R' = R_x$, then we add the new equivalences Eq_{new} , created by the coinductive procedure, to the set of circularities. Thus we reuse the already proven equivalences in future proofs.

The derivatives of the initial ERE R_0 with respect to all events in the alphabet A are generated in a depth first fashion. The pseudo code for the whole algorithm is given in Figure 1.

```

dfs( $R$ )
begin
  foreach  $x \in A$  do
     $R_x \leftarrow R\{x\}$ ;
    if  $\exists R' \in S$  such that  $\mathcal{B} \cup Eq_{all} \cup Eq_{new} \Vdash R' = R_x$  then
       $\delta(R, x) = R'$ ;  $Eq_{all} \leftarrow Eq_{all} \cup Eq_{new}$ 
    else  $S \leftarrow S \cup \{R_x\}$ ;  $\delta(R, x) = R_x$ ; dfs( $R_x$ ); fi
  endfor
end

```

Fig. 1. ERE to minimal DFA generation algorithm

In the procedure **dfs** the set of final states F consists of the EREs from S which contain ϵ . This can be tested efficiently using the equations (10-16) in Subsection 2.2. The DFA generated by the procedure **dfs** may now contain some states which are non-final and from which the DFA can never reach a final state. We remove these redundant states by doing a breadth first search in backward direction from the final states. This can be done in time linear in the size of the DFA.

Theorem 4.1 *If D is the DFA generated for a given ERE R by the above algorithm then*

- 1) $\mathcal{L}(D) = \mathcal{L}(R)$,
- 2) D is the minimal DFA accepting $\mathcal{L}(R)$.

Proof:

- 1) Suppose $a_1a_2\dots a_n \in \mathcal{L}(R)$. Then $\epsilon \in R\{a_1\}\{a_2\}\dots\{a_n\}$. Let $R_i = R\{a_1\}\{a_2\}\dots\{a_i\}$; then $R_{i+1} = R_i\{a_{i+1}\}$. To prove that $a_1a_2\dots a_n \in \mathcal{L}(D)$, we use induction to show that for each $1 \leq i \leq n$, $R_i \equiv \delta(R, a_1a_2\dots a_i)$. For the base case if $R_1 \equiv R\{a_1\}$ then **dfs** extends the transition function by setting $\delta(R, a_1) = R$. Therefore, $R_1 \equiv R = \delta(R, a_1)$. If $R_1 \neq R$ then **dfs** extends δ by setting $\delta(R, a_1) = R_1$. So $R_1 \equiv \delta(R, a_1)$ holds in this case also. For the induction step let us assume that $R_i \equiv R' = \delta(R, a_1a_2\dots a_i)$. If $\delta(R', a_{i+1}) = R''$ then from the **dfs** procedure we can see that $R'' \equiv R'\{a_{i+1}\}$. However, $R_i\{a_{i+1}\} \equiv R'\{a_{i+1}\}$. So $R_{i+1} \equiv R'' = \delta(R', a_{i+1}) = \delta(R, a_1a_2\dots a_{i+1})$. Also notice $\epsilon \in R_n \equiv \delta(R, a_1a_2\dots a_n)$; this implies that $\delta(R, a_1a_2\dots a_n)$ is a final state and hence $a_1a_2\dots a_n \in \mathcal{L}(D)$.

Now suppose $a_1a_2\dots a_n \in \mathcal{L}(D)$. The proof that $a_1a_2\dots a_n \in \mathcal{L}(R)$ goes in a similar way by showing that $R_i \equiv \delta(R, a_1, a_2\dots a_i)$.

- 2) If DFA D is not minimal then there exists at least two states p and q in D such that p and q are *equivalent* [15] i.e. $\forall w \in A^* : \delta(p, w) \in F$ iff $\delta(q, w) \in F$, where F is the set of final states. This means, if R_1 and R_2 are the EREs associated with p and q respectively in **dfs** then $\mathcal{L}(R_1) = \mathcal{L}(R_2)$ i.e. $R_1 \equiv R_2$. But **dfs** ensures that no two EREs representing the states of the DFA are equivalent. So we get a contradiction.

□

5 Implementation and Evaluation

We have implemented the coinductive rewriting engine in the rewriting specification language Maude 2.0 [4]. The interested readers can download the implementation from the website <http://fsl.cs.uiuc.edu/rv/>. The operations on extended regular languages that are supported by our implementation are \sim for negation, $_ * _$ for Kleene Closure, $_ _$ for concatenation, $_ \& _$ for intersection, and $_ + _$ for union in increasing order of precedence. Here, the intersection operator $_ \& _$ is a syntactic sugar and is translated to an ERE containing union and negation using De Morgan's Law:

$$\text{eq } R1 \& R2 = \sim (\sim R1 + \sim R2) .$$

To evaluate the performance of the algorithm we have generated the minimal DFA for all possible EREs of size up to 9. Surprisingly, the size of any DFA for EREs of size up to 9 did not exceed 9. Here the number of states gives the

size of a DFA. The following table shows the performance of our procedure for the worst EREs of a given size. The code is executed on a Pentium 4 2.4GHz, 4 GB RAM linux machine.

Size	ERE	no. of states in DFA	Time (ms)	Rewrites
4	$\neg(a b)$	4	< 1	863
5	$(a \neg b)^*$	4	< 1	1370
6	$\neg((a \neg b)^*)$	4	1	1453
7	$\neg(a \neg a a)$	6	1	2261
8	$\neg((a \neg b)^* b)$	7	1	3778
9	$\neg(a \neg a b) b$	9	5	9717

Example 5.1 In particular, for the ERE $\neg(a \neg a b) b$ the generated minimal DFA is given in Figure 2.

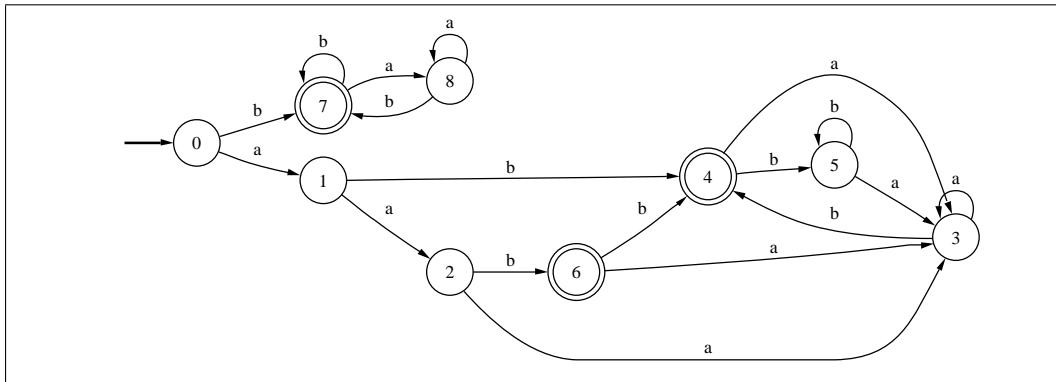
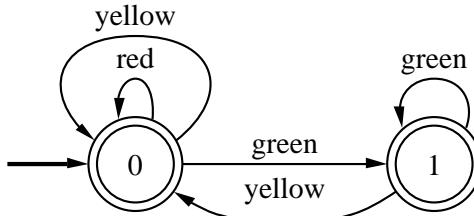


Fig. 2. $\neg(a \neg a b) b$

Example 5.2 The ERE $\neg((\neg \text{empty}) (\text{green red}) (\neg \text{empty}))$ states the safety property that it should not be the case that in any trace of a traffic light we see green and red consecutively at any point. The set of events are assumed to be { green, red, yellow}. We think that this is the most intuitive and natural expression for this safety property. The implementation took 1ms and 1663 rewrites to generate the minimal DFA with 2 states. The DFA is given in Figure 3.

However for large EREs the algorithm may take a long time to generate a minimal DFA. The size of the generated DFA may grow non-elementarily in the worst case. We generated DFAs for some complex EREs of larger sizes and got relatively promising results. One such sample result is as follows.

Example 5.3 Let us consider the following ERE of size 110

Fig. 3. $\neg ((\neg \text{empty}) (\text{green red}) (\neg \text{empty}))$

$$\begin{aligned}
 & (\neg \$)^* \$ (\neg \$)^* \cap \\
 & (0 + 1 + \#)^* \# (\\
 & ((0 + 1)0 \# (0 + 1 + \#)^* \$ (0 + 1)0 + (0 + 1)1 \# (0 + 1 + \#)^* \$ (0 + 1)1) \\
 & \cap (0(0 + 1) \# (0 + 1 + \#)^* \$ 0(0 + 1) + 1(0 + 1) \# (0 + 1 + \#)^* \$ 1(0 + 1))).
 \end{aligned}$$

This ERE accepts the language L_2 , where

$$L_k = \{\sigma \# w \# \sigma' \$ w \mid w \in \{0, 1\}^k \text{ and } \sigma, \sigma' \in \{0, 1, \#\}^*\}$$

The language L_k was first introduced in [3] to show the power of alternation, used in [27] to show an exponential lower bound on ERE monitoring, and in [19,20] to show the lower bounds for model checking. Our implementation took almost 18 minutes to generate the minimal DFA of size 107 and in the process it performed 1,374,089,220 rewrites.

The above example shows that the procedure can take a large amount of time and space to generate DFAs for large EREs. To avoid the computation associated with the generation of minimal DFA we plan to maintain a database of EREs and their corresponding minimal DFAs on the internet. Whenever someone wants to generate the minimal DFA for a given ERE he/she can look up the internet database for the minimal DFA. If the ERE and the corresponding DFA exists in the database he/she can retrieve the corresponding DFA and use it as a monitor. Otherwise, he/she can generate the minimal DFA for the ERE and submit it to the internet database to create a new entry. The database will check the equivalence of the submitted ERE and the corresponding minimal DFA and insert it in the database. In this way one can avoid the computation of generating minimal DFA if it is already done by someone else. To further reduce the computation, circularities could also be stored in the database.

5.1 Online Monitor Generation and Visualization

We have extended our implementation to create an internet server for optimal monitor generation that can be accessed from the url <http://fsl.cs.uiuc.edu/rv/>. Given an ERE the server generates the optimal DFA monitor for a user. The user submits the ERE through a web based

form. A CGI script handling the web form takes the submitted ERE as an input, invokes the Maude implementation to generate the minimal DFA, and presents it to the user either as a graphical or a textual representation. To generate the graphical representation of the DFA we are currently using the GraphViz tool [6].

6 Conclusion and Future Work

We presented a new technique to generate optimal monitors for extended regular expressions, which avoids the traditional technique based on complementation of automata, that we think is quite complex and not necessary. Instead, we have considered the (co)algebraic definition of EREs and applied coinductive inferencing techniques in an innovative way to generate the minimal DFA. Our approach to store already proven equivalences has resulted into a very efficient and straightforward algorithm to generate minimal DFA. We have evaluated our implementation on several hundreds EREs and have got promising results in terms of running time. Finally, we have installed a server on the internet which can generate the optimal DFA for a given ERE.

At least two major contributions have been made. Firstly, we have shown that coinduction is a viable and quite practical method to prove equivalence of extended regular expressions. Previously this was done only for regular expressions without complementation. Secondly, building on the coinductive technique, we have devised an algorithm to generate minimal DFAs from EREs. At present we have no bound for the size of the optimal DFA, but we know for sure that the DFAs we generate are indeed optimal. However we know that the size of an optimal DFA is bounded by some exponential in the size of the ERE. As future work, it seems interesting to investigate the size of minimal DFAs generated from EREs, and also to apply our coinductive techniques to generate monitors for other logics, such as temporal logics.

References

- [1] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
- [2] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1):51–72, 1995.
- [3] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.
- [4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In *3rd International Workshop on Rewriting Logic and its Applications (WRLA '00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

- [5] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
- [6] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(1):1203–1233, September 2000.
- [7] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
- [8] J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proceedings, Automated Software Engineering '00*, pages 123–131. IEEE, 2000. (Grenoble, France).
- [9] J. Goguen, K. Lin, and G. Rosu. Conditional circular coinductive rewriting with case analysis. In *Recent Trends in Algebraic Development Techniques (WADT'02)*, Lecture Notes in Computer Science, to appear, Frauenschiemsee, Germany, September 2002. Springer-Verlag.
- [10] K. Havelund and G. Roşu. Java PathExplorer – A Runtime Verification Tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, Montreal, Canada, June 18 - 21, 2001.
- [11] K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
- [12] K. Havelund and G. Roşu. *Runtime Verification 2002*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2002. Proceedings of a *Computer Aided Verification (CAV'02)* satellite workshop.
- [13] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
- [14] S. Hirst. A new algorithm solving membership of extended regular expressions. Technical report, The University of Sydney, 1989.
- [15] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
- [16] L. Ilie, B. Shan, and S. Yu. Fast algorithms for extended regular expression matching and searching. In H. Alt and M. Habib, editors, *Proceedings of the 20th International Symposium on Theoretical Aspects of Computer (STACS 03)*, volume 2607 of *Lecture Notes in Computer Science*, page 179. Springer-Verlag, Berlin, 2003.

- [17] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
- [18] J. Knight and E. Myers. Super-pattern matching. *Algorithmica*, 13(1/2):211–243, 1995.
- [19] O. Kupferman and M. Y. Vardi. Freedom, Weakness, and Determinism: From linear-time to branching-time. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 81–92, 1998.
- [20] O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Proceedings of the Conference on Computer-Aided Verification*, 1999.
- [21] O. Kupferman and S. Zuhovitzky. An Improved Algorithm for the Membership Problem for Extended Regular Expressions. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, 2002.
- [22] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
- [23] G. Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(4):430–448, 1992.
- [24] T. O’Malley, D. Richardson, and L. Dillon. Efficient Specification-Based Oracles for Critical Systems. In *In Proceedings of the California Software Symposium*, 1996.
- [25] D. J. Richardson, S. L. Aha, and T. O. O’Malley. Specification-Based Test Oracles for Reactive Systems. In *Proceedings of the Fourteenth International Conference on Software Engineering, Melbourne, Australia*, pages 105–118, 1992.
- [26] G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.
- [27] G. Roşu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In *Rewriting Techniques and Applications (RTA ’03)*, Lecture Notes in Computer Science, to appear. Springer-Verlag, 2003.
- [28] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR 98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer-Verlag, 1998.
- [29] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. Technical Report UIUCDCS-R-2003-2334, University of Illinois at Urbana Champaign, April 2003.

- [30] B. Watson. A taxonomy of deterministic finite automata minimization algorithms. Technical Report ISSN 0926-4515, Eindhoven University of Technology, Eindhoven, The Netherlands, 1993.
- [31] H. Yamamoto. An automata-based recognition algorithm for semi-extended regular expressions. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, pages 699–708, 2000.

Appendix Z:

Modeling A System Of UAVs On A Mission

Predrag Totic*, Myeong-wuk Jang, Smitha Reddy, Joshua Chia,
Liping Chen, Gul Agha

Open Systems Laboratory, Department of Computer Science

University of Illinois at Urbana-Champaign

{p-totic, mjang, sreddy1, jimchia, lchen2, agha}@cs.uiuc.edu

Abstract

We outline a parametric model of a system of **unmanned aerial vehicles (UAVs)** on a mission. The **UAVs** have to accomplish their mission composed of several tasks as efficiently as possible, while satisfying a heterogeneous set of physical and communication constraints. **UAVs** can be viewed as an example of a highly dynamic **multi-agent system (MAS)**. These **UAVs** may be required to autonomously make decisions, communicate, coordinate, adapt to rapidly changing environments and efficiently perform their tasks in real-time and under the limitations of local, incomplete and/or noisy knowledge of their surroundings. In particular, an individual **UAV** in our work can be viewed as an **agent**: it is autonomous, goal-driven, can affect and be affected by its environment, has its own behavior strategy, can communicate with its peers, and may find it beneficial to cooperate and coordinate not only to avoid collisions, but also in order to accomplish its set of tasks more effectively. We focus herein on two aspects of agent-based modeling of **UAVs**: modeling autonomous decision-making of the individual vehicles viewed as autonomous agents, and different models of **UAV** coordination.

Keywords: *unmanned aerial vehicles, agent-based modeling, agent coordination, agent autonomy*

1 Introduction & Motivation

A collection of **Unmanned Aerial Vehicles (UAVs)** on a mission provides an ideal framework for identifying, modeling and analyzing many interesting paradigms, design parameters and solution strategies applicable not only to autonomous unmanned vehicles, but to **Multi-Agent Systems (MAS)** in general. UAVs are finding their application in a variety of contexts, e.g., they are being increasingly used for various surveillance, reconnaissance, and target-and-rescue missions. These UAVs carry sophisticated payloads, as they are designed to accomplish increasingly complex, multi-task missions. In particular, a typical UAV is equipped with certain *sensors* such as, e.g., radars. With these sensors, each UAV probes its environment and forms a (local) “picture of the world” on which its future actions may need to be based. A UAV is also equipped with some *communication capabilities*, that enable it to communicate with other UAVs and/or

the ground or satellite control. This communication enables a UAV to have an access to the information that is not local to it - that is, the information not directly accessible to UAV’s sensors.

While trying to accomplish their mission (typically, a set of pre-defined and/or dynamically arising tasks as in the examples above), these UAVs need to respect a heterogeneous set of constraints on their physical and communication resources. The UAVs also need to be able to communicate and cooperate with each other. Their cooperation can range from merely assuring that they stay out of each other’s way (collision avoidance) to enabling themselves to adaptively and dynamically divide-and-conquer their tasks. This latter, higher form of cooperation (coordination) we also call *goal-driven* cooperation (respectively, coordination).

Not all kinds of UAVs can be reasonably considered *agents*; e.g., those that are remotely controlled

*Contact author

throughout their mission would not qualify for (*autonomous*) *agents* in the usual sense. However, for the reasons of system scalability, dependability and robustness, increasingly complex and autonomous unmanned vehicles are being studied, designed and manufactured. We are interested in UAVs that are not remotely controlled and that have the ability to make their own decisions in real time. This ability of autonomous decision making would “qualify” UAVs to be considered *autonomous agents* in the usual, computer science sense of the word.¹ We are also assuming either no central control, or only a limited central control. In particular, the knowledge of the world that each UAV possesses is, in general, assumed to be *local*, possibly *noisy*, to vary with time, and to be augmentable, at a certain cost, via communication with other UAVs.

Some of the problems that have been extensively studied in the context of UAVs include motion planning and conflict detection and resolution (see, e.g., [BIC], [KUC], or [PAL]). What has drawn considerably less attention is modeling and analysis of the task-driven, goal-oriented behaviors of UAVs viewed as agents. Herein, we focus on some critical agent-based modeling paradigms applied to UAVs, namely, models of agent-to-agent coordination and the individual agent autonomy.

The rest of the paper is organized as follows. In §2, we give a high-level problem formulation, introduce some terminology, and reflect on some of the main assumptions of our modeling framework. The central part of the paper, section §3, is dedicated to identifying and discussing some of the most crucial design parameters of the model - those pertaining to modeling tasks, and UAV coordination and autonomous decision-making. We outline some possible extensions of the modeling framework in §4, and briefly summarize in §5.

2 Problem Formulation

In this section, we first briefly discuss the model at a high level: what are the UAVs trying to accomplish, both individually and as a single multi-agent system with a common goal, how we model these goals, and how we model UAVs strategies and mechanisms for accomplishing their goals. We also introduce the necessary terminology along the way. We conclude the section with a brief discussion about the main assump-

tions made in our model, and some of their implications.

A collection of N UAVs needs to accomplish a certain complex mission - such as any combination of surveillance, reconnaissance, target detection and/or target identification. We model this mission with a collection of M *interest points* (*IPs*). An interest point is a semantic extension of the more common notion of a *target* - in addition to targets proper, an IP may also refer to, e.g., a small local region of interest, that may or may not include “real targets”, but is nonetheless worth while exploring. Each interest point j has a dynamically changing value associated with it, $\Pi_j(t)$. An IP may be static or mobile. A mobile IP j , at any time step t , is completely and uniquely specified by its position and velocity vectors, $\psi_j(t)$ and $\xi_j(t)$, respectively, and its value $\Pi_j(t)$. A UAV V_i is driven by the desire to increase its own utility, U_i , by consuming as much of value of various IPs as possible. The total amount of value is assumed to be bounded at all times. Consequently, the UAVs may end up competing for this limited resource. The *coordination model* describes how much UAVs cooperate and divide-and-conquer the IPs in order to function efficiently as a system.

At one extreme, if there is no coordination, each UAV acts entirely autonomously and, assuming no central control or other outside mechanisms, *selfishly*. At another extreme, in the leader-based coordination models, the UAVs that are not leaders obey their respective leaders, thereby sacrificing (temporarily or permanently) their autonomy as agents. Assuming unbounded communication radius for the UAV-to-UAV communication, a single-leader model becomes equivalent to a centralized model, with a possibly incomplete and/or noisy knowledge of the “world”. If, however, there are several leaders and different groups of “followers” associated with each leader, and if the radius of communication is non-trivially finite, then one may expect to encounter many of the fundamental challenges in distributed computation and communication, such as dynamic leader election and group formation problems, distributed consensus reaching, limits to local individual or group knowledge and their implications, and other issues (see, e.g., [TEL]).

Thus, from an individual UAV’s perspective, the goal is to maximize its own utility, by visiting as many interest points and consuming as much of their value as possible. This is accomplished by following a certain

¹We leave aside the fact that there is no general agreement within the agent research community on *what exactly qualifies an entity* (such as a computer program, or a UAV with its sensors, effectors and software) *to be considered an agent* [FRA].

either fixed or dynamically changing (adaptable) *individual behavior strategy*. This strategy can be specified by an appropriate *individual behavior function*, Θ_i , that UAV V_i follows as long as there is no outside signal telling the UAV it should start doing something else. An example of such outside signal is a request to a given UAV to join a newly formed group; if such request comes from a leader whose supremacy in authority is recognized, the follower UAV will have to abandon its current behavior and comply with the leader's desires, thereby, in a sense, giving up its individual autonomy.

From a system's perspective, on the other hand, it is the successfulness of the entire mission that matters - not the gratification of individual agents. How to translate the individual utilities into the global utility maximization in the framework where, in general, both cooperation and competition are to be expected, is a challenging *incentive engineering* problem [CAN]. We address these issues elsewhere, where our constraint optimization based framework for modeling UAV missions is the central theme [TOS].

We conclude this section with stating some basic assumptions made both in the agent-based UAV model outlined herein, and in our simulation platform based on this model². One important assumption pertains to the nature of *time*. First, the time steps are discrete. Second, we assume the existence of the *global clock* and, therefore, *global time*. The *global clock* could be provided, say, by a central satellite-based control, and we also assume that all UAVs at any time have an instantaneous access to this global clock. The assumption about global time is (tacitly) made in most of the work on *MAS*, where the existence of a global clock is taken for granted. Without it, modeling and analysis of UAV-like distributed systems becomes considerably more difficult. The UAVs are assumed to communicate with one another (or, when applicable, with the central control) exclusively via *message passing*. We also assume that all communication is perfectly *synchronous*. It is well-known that the more realistic assumption of asynchronous communication renders many important distributed coordination and agreement problems formally undecidable [TEL].

3 Modeling UAVs' Tasks, Coordination and Autonomy

We now discuss in some detail what we consider to be the most critical design parameters in our agent-based model of UAVs on a multi-task mission. The parameters of interest include ratio of the number of UAVs to the number of IPs, sensor ranges, communication ranges, a choice of a coordination model and strategy, a model of UAV's individual behavior strategy (or, in heterogeneous scenarios where UAVs are distinguishable, *strategies*), models of both individual and system *adaptability*³, and a choice of the model of UAVs' knowledge of their environment - such as whether this knowledge is local or global, perfect or noisy, etc.

Due to space limitations, we focus herein on three issues: quantitative models of interest points, models of individual UAV's autonomy, and UAV coordination models.

3.1 A Simple Model of UAV Tasks

Our model of UAVs on a mission emphasizes the goal-orientedness of UAVs as agents. UAVs are not merely flying around and trying to avoid colliding with one another or with other obstacles, but are actually trying to accomplish some set of tasks. Thus our model is trying to capture paradigms beyond the usual motion planning and obstacle avoidance problems. Mathematically, rather than having merely to solve an instance of *Distributed Constraint Satisfaction (DCS)* problem⁴ [YOK], our UAVs have, in addition to obeying a number of physical and communication constraints, also an *objective* or *utility function* that they strive to maximize. We discuss some possible ways of translating this desire to maximize utility into UAV's individual decision-making strategies in the next section. First, however, we outline our quantitative model of UAVs' tasks.

As our goal is to capture UAVs acting in possibly heterogeneous and highly dynamic environments, where not all tasks need be (i) identical, or (ii) known ahead of time, and where, in general, no central control is available to provide each UAV with the information about each of the tasks, a natural starting point in our quest to quantify the mission successfulness is to specify a simple *quantitative* notion of a *task*. We also need

²For more on our UAV simulation and some experimental results, see [JAN].

³By *adaptability* we mean the ability to change the individual strategies and coordination models based on observed changes in the environment, including but not limited to any form of a *feedback*, such as an appropriate *payoff*, received from the environment.

⁴DCS by itself is, in general, computationally intractable, as even its centralized version, being a nontrivial generalization of the well-known *Satisfiability problem*, is **NP-hard**.

a simple model for possible heterogeneity of different tasks. The basic task to be serviced in our model is an *interest point*.

Since not all tasks, or interest points, are necessarily the same, one simple way to capture this heterogeneity is to assign a time-dependent value function, $\Pi_j(t)$, to each interest point j , $1 \leq j \leq M$. When a UAV discovers an IP, it gets attracted by its value. Assuming a UAV is aware of two or more IPs at a given time step (by either having sensed those IPs, or because some other UAV has broadcast the IPs' locations and (estimated) values), the UAV needs to decide which IP is currently *most attractive* to it. The exception to this general rule is when, in task-driven coordination models, the UAV gets instructions from another UAV or the ground or satellite control what it is supposed to do next.

When a UAV arrives to an IP's location (or within a specified small distance from it), it starts consuming its value, thereby increasing its own utility or payoff. This value is consumed at some rate, d . In our simulations (see [JAN]), d was assumed to be a constant; in general, various models where d may depend on time, UAV's index i , and/or IP's index j , are worth considering.

To illustrate the general usefulness of the concept of IPs and their *value functions*, we sketch two special cases as examples.

First, let's assume that UAVs are on a surveillance mission. That is, each IP (or a set of IPs) needs to be revisited repeatedly. Some regions (represented by IPs) may be so important that they require presence (i.e., one or more UAVs essentially hovering or circling in their vicinity) at all times. The simple way to represent that in our model is to make the function $\Pi_j(t)$ independent of time: even though some UAVs are "consuming" the value of such an IP, while those UAVs' utility is increasing, the IP's value actually stay the same, thereby assuring that those IPs keep their attractiveness. If some IPs need to be revisited periodically but do not require ceaseless surveillance, then the function $\Pi_j(t)$ of such an IP j can be made periodic: once a UAV arrives to j , the value starts going down until UAV leaves; after some number of time steps, the value jumps back up again, thereby making j (more) attractive again, and thus prompting the UAVs to come back to this IP.

The second example is to consider an IP that is an actual target. Once its location has been discovered, one or more UAVs approach this target. Once a particular UAV, V_i , gets within some pre-specified

distance from the target, with probability p , the UAV consumes the target's entire value at once, with probability p_i . That is, Π_j of this IP goes to zero in one time step, and the UAV's payoff increases accordingly with probability p_i , whereas, with probability $1 - p_i$, V_i "misses" the target, so that the value Π_j remains unaltered. Whether the UAV V_i stays "at the target" j for one time step irrespective of the outcome, or for as many time steps as is needed for a success to occur, are different possibilities that can be modeled with different choices of the UAV's individual behavior functions.

What are, then, the general properties of function Π , and what parameters does it depend upon? At time step $t + 1$, the value $\Pi_j(t + 1)$ of IP j can be reasonably expected to depend on the value at the previous time step, $\Pi_j(t)$, the number of UAVs servicing this IP at time t , that we denote by $n_j(t)$, and the value consumption rate, d . In addition, the value function may explicitly depend on time. For instance, in the surveillance example above, the parameters $\Pi_j(t)$, d and $n_j(t)$ alone cannot capture a jump in $\Pi_j(t + 1)$ value due to some form of *aging*. Thus, the general form of the IP value functions we are interested in can be written as

$$\Pi_j(t + 1) = F(\Pi_j(t), n_j(t), d_{i,j}(t), t)$$

for some integer-valued or real-valued function F . A particular choice of Π that we have extensively experimented with [JAN] is

$$\Pi_j^*(t + 1) = \max\{\Pi_j^*(t) - d \cdot n_j(t), 0\},$$

where d is an integer constant. While this particular IP value function is always nonnegative, we also consider $\Pi(t)$ that can be negative; such value functions are useful whenever one wishes to model certain regions that UAVs should strive to avoid - such as, e.g., dangerous regions in the mission area of little actual value. Similarly, not all IP value functions need be nonincreasing in time like Π_j^* is; depending on what kind of IPs are modeled, value functions may be chosen to be, e.g., periodic or nondecreasing in time, and the like.

3.2 Models of UAV Autonomy

In order for any type of *intelligent vehicles* to be considered *autonomous agents*, they have to be capable of *autonomous decision making* without direct assistance of a human or other outside operator. We outline a simple model of autonomy applicable to UAVs that would render UAVs proper autonomous agents, albeit of a perhaps fairly restricted kind. UAVs are

goal-driven entities. They fulfill their goals by servicing tasks. In our modeling framework, tasks are given as interest points and UAVs, loosely speaking, strive to consume as much of interest points' *value* as fast as possible. As we assume that a single UAV can consume value from at most one IP at any single time step, the question arises: among several candidate IPs, how should a UAV choose in what order it is to visit these IPs? Therefore, it can be argued that each UAV faces an *online scheduling problem*. For simplicity, we consider a simplified version of dynamic online scheduling, and only ask, given a set of interest points whose current positions and (estimated) values are known to a particular UAV, which IP among those points should the UAV select to visit first?

We model the individual UAV's autonomous decision-making with UAV's *individual behavior functions*, Θ_i . Given a set of IPs with their current positions and values⁵, a UAV V_i evaluates its behavior function Θ_i that returns the index j^* of the IP that, if the UAV selects that IP as its next task to service, this choice is expected or to maximize the *estimated increase* in UAV's *utility*. Therefore, each UAV is assumed to behave *greedily*. However, a great variety of greedy strategies can be specified via different choices of the functions Θ_i .

Some variables that individual behavior functions can be expected to depend on are the UAV's distance from the given IP, the IP's current value (or its estimate), and the estimated competition for that IP and its value - viz., the number of other UAVs in the IP's vicinity. Let $\psi_j(t)$ be the position of IP j at time t , and let $n_{j,r}$ be the total number of UAVs within the distance r from IP j . Then one class of models of the i -th UAV's target selection strategy is specified by

$$\Theta_i(t) = \arg\{ \max_{1 \leq j \leq M} G(\Pi_j, \|x_i - \psi_j\|, n_{j,r}, t) \}$$

where G is an integer-valued function that is increasing in Π_j and nonincreasing in distance of the UAV from the IP j given by $\|x_i(t) - \psi_j(t)\|$. This function specifies what IP should UAV V_i pick as the estimated short-term optimal choice. Notice that, for simplicity, *relative velocity* of a UAV with respect to the interest point is not taken into account. One example of a simple greedy individual behavior that fits the given description is

$$\Theta_i(t) = \arg\{ \max_{1 \leq j \leq M} [\frac{\Pi_j(t) - n_{j,r}(t) \cdot d}{\|x_i(t) - \psi_j(t)\|}] \},$$

where it is assumed that the minimal distance of any UAV from any IP is strictly positive.

Everything said thus far about UAVs' individual behavior strategies rests on the assumption that each UAV acts strictly selfishly, and largely independently (except for the dependence of Θ_i on $n_{j,r}$) from what other UAVs do. Once UAV-to-UAV communication and coordination are taken into account, modeling UAV's autonomous agent behavior becomes more complex. In particular, in addition to the already mentioned parameters, each $\Theta_i(t)$ would be expected to also depend on *the set of messages that i -th UAV has received from other UAVs* at time steps $t' \leq t$. We discuss some models of UAV coordination next.

3.3 Models of UAV Coordination

We now discuss some possible design choices for modeling UAV-to-UAV coordination.

At one extreme, a single UAV becomes the "group leader", and this leader then broadcasts to other UAVs what it wants them to do. Typically, the leader is the *first* UAV that detects one or more IPs in a particular region of the *mission area*. In case of a tie (where two or more UAVs announce their claim to leadership simultaneously), the tie is broken according to some pre-specified rule (e.g., the UAV with the lowest index wins). Assuming the UAV-to-UAV communication radius is infinite⁶, and if the bandwidth availability is not an issue, this, single-leader scenario is very similar to a centralized control model. The one difference is that the leader's knowledge about the environment, in general, can be expected to be incomplete and/or imperfect (i.e., noisy), and that this knowledge is likely to dynamically change in often unpredictable ways.

While the single-leader model is perhaps the simplest to analyze and relatively easy to simulate, it also suffers from a number of shortcomings. These shortcomings can be divided into two general categories. One category are the usual problems with centralized or quasi-centralized control models, such as "ungraceful" degradation (due to a single point of failure), and the possible communication bottleneck at the leader node. The second category of potential troubles is peculiar to any situation where a single leader is itself "just another agent", whose sensors or communication links could be unreliable, whose local and possibly noisy "picture of the world" is imposed onto everyone else even though perhaps other agents have more accurate knowledge or more reliable links, and the like. Any

⁵In case of *mobile targets*, current velocities would be also needed.

⁶The infinite communication range assumption applies whenever this range is unbounded for all practical purposes - which is the case if, for example, the diameter of the entire mission area is less than the diameter of the UAV-to-UAV communication range.

satisfactory solution, therefore, has to provide mechanisms for UAVs not only to “talk back” to their leaders, but also for the *ad hoc network* of UAVs to be able to detect possible troubles with the leader, and, if need be, dynamically reconfigure itself and elect a new leader.

At the other extreme, we consider models where there is no explicit task-driven coordination. In these models, not only is the whole group of UAVs autonomous, but also each individual UAV within this system acts as an entirely autonomous agent. In other words, each UAV simply follows its own strategy by recomputing its *individual behavior function*, irrespective of what others do. This individual behavior function is the agent’s strategy for maximizing its own individual payoff. The UAVs may still wish to communicate with one another, but there is no explicit coordination as to how to accomplish the mission more efficiently, how to divide-and-conquer tasks, etc. While, in the context of individual utility maximization this situation may be considered a default scenario, we view it as an extreme in the more appropriate, *joint utility* framework, where all UAVs have a single mission to accomplish, and where the successfulness of the entire system in accomplishing that mission - rather than that of the individual vehicles - is what matters [TOS]. Thus the “no goal-driven cooperation” scenario can serve as a yard stick with respect to which the effectiveness of various coordination strategies can be measured.

In between the two tentative extremes - the leader-based coordination on one, and the no explicit coordination model on the other hand - are many intermediate cases, and many possible coordination strategies. These intermediate coordination models are, typically, more flexible but also more complex than the leader-based approaches. We refer to this broad class of *intermediate* coordination strategies as *leaderless coordination models*.⁷

Let us summarize regarding the possible choices of a coordination model, and the tradeoffs these choices entail. Without any explicit goal-driven coordination, each UAV follows its own, pre-specified (but possibly adaptable) individual behavior strategy. The parameters to this strategy, encapsulated in each UAV’s *individual behavior function*, are provided by the environment, i.e. by the (possibly noisy) data about the environment gathered by the vehicle’s sensors, by the UAV’s knowledge of its “internal state”, and by the communication with other UAVs and/or the ground control that helps the UAV navigate, detect and avoid

possible collisions, ant the like. Goal-driven coordination, then, entails different UAVs occasionally having to, temporarily or permanently, *abandon their individual behavior strategies*, and to begin doing what they have reached an agreement with other UAVs they ought to be doing. In general, coordination should aid the system to do better, and therefore any reasonable coordination strategy should be expected, given the same mission and the same set of tasks and other environment parameters, to perform at least as well as the corresponding coordination-free, purely autonomous individual behavior strategy. This intuition has been confirmed in some restricted scenarios that we have simulated [JAN]. We briefly argue in favor of a broader validity of our claim regarding the expected benefits of the *goal-driven coordination*. Assuming perfectly reliable communication and agent’s perfect knowledge of the environment - two typically unrealistic assumptions in practice - any model of goal-oriented coordination implies, at the very least, some *knowledge sharing*, i.e., more information available to each UAV than what is provided by the UAV’s sensors and communication about possible conflicts *alone*. More information, on the other hand (and leaving aside for the moment the issues of communication, storage and processing overheads), should not make the system or its components do any worse than without that additional information. Once the assumptions of perfect knowledge about the IPs and perfect communication links are dropped, however, the problem of choosing the right coordination strategy becomes both intuitively and analytically overwhelming, and therefore our view is that, in that case, there is no substitute for computer simulation and intense experimentation in various scenarios.

We also point out that, irrespective of the assumptions about reliability of communication links and UAVs’ sensors, any coordination *necessarily* requires more communication - and, in the real world, communication does not come for free. It also means that the agents have to execute an appropriate coordination mechanism, which may mean a considerable computational overhead - possibly prohibitively costly in a real-time application. Thus, in general, one can expect a tradeoff between the amount and nature of coordination, and the extra cost of this coordination. Hence, for example, in those scenarios where goal-driven coordination is expected or experimentally shown to be of a little benefit, by the Occam’s Razor principle, the simpler and less costly strategies with no explicit coordination would likely be preferable.

⁷Our UAV simulator has an implementation of a variant of each of these three categories of coordination models [JAN].

4 Some Future Plans

We now outline some possible extensions of the agent-based modeling framework presented in the previous sections. Regarding the nature of tasks, in addition to the interest points' values, we consider introducing different IP *types*. This naturally extends the model so that it can capture more heterogeneous scenarios, where both UAVs and IPs are *distinguishable* and, in particular, the UAVs become specialized: a UAV can handle only those IPs that are of a compatible type.

An important remark, leading to a considerable model extension, is that currently, for each IP j , its Π_j is assumed to represent the j 's true (i.e., objective) value, irrespective of which UAV may see IP j , and from how far away. That is, the tacit assumption is that, once an IP is discovered by the UAVs, all the UAVs who are aware of this IP's existence immediately also know its *exact* current value. This *perfect knowledge* assumption can be appropriately relaxed for the sake of the model's realism. The more reasonable assumption is that each UAV has its own, *local* and *imperfect* estimate of Π_j of those IPs j that the UAV knows about. In case of imperfect and/or incomplete local knowledge of the tasks, each UAV's individual behavior function, Θ_i , naturally becomes probabilistic, i.e., UAVs are required to make decisions in the presence of uncertainty. In particular, Θ_i is now a function of an *estimated attractiveness* of each IP j that UAV V_i is aware of, rather than of the exact actual value Π_j as this actual value, in general, need not be known to V_i . Ability of decision-making under uncertainty is often considered one of the hallmarks of most autonomous agents one finds in the literature [PAR].

5 Summary

We have presented an agent-based approach to modeling UAVs on a mission made of multiple tasks. Assuming no remote control, the UAVs share many properties characteristic of autonomous agents, such as goal-orientedness, pro-activeness, the ability to affect and be affected by the environment, autonomous decision-making under uncertainty, and peer-to-peer communication, coordination and cooperation. The current emphasis of our modeling is on some simple yet interesting classes of autonomous agent behavior strategies and of goal-driven agent coordination. It is our hope that

future extensions and improvements of the modeling framework presented herewith, and simulations guided by similar models, will increase the understanding and enhance future design of both autonomous UAVs and other intelligent vehicles, and multi-agent systems in general.

Acknowledgment: Work presented herewith was supported by the **DARPA IPTO TASK Program**, contract number **F30602-00-2-0586**.

Bibliography

[BIC] A. Bicchi, L. Pallottino, "On optimal cooperative conflict resolution for air traffic management systems", IEEE Trans. Intelligent Transportation Systems, vol. 1, Dec. 2000

[CAN] D. H. Cansever, "Incentive Control Strategies For Decision Problems With Parametric Uncertainties", Ph.D. thesis, Univ. of Illinois Urbana-Champaign, 1985

[FRA] S. Franklin, A. Graesser, "Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents", Proc. 3rd Int'l Workshop on Agent Theories, Architectures & Languages, Springer-Verlag, 1996

[JAN] M. Jang, S. Reddy, P. Tosic, L. Chen, G. Agha, "An Actor-based Simulation for Modeling Coordination among UAVs", submitted to 15th Euro. Simulation Symposium & Exhibition, Delft, The Netherlands, 2003

[KUC] J. K. Kuchar, L. C. Yang, "A review of conflict detection and resolution modeling methods", IEEE Trans. Intelligent Transport. Systems, vol. 1, Dec. 2000

[PAL] L. Pallottino, E. M. Feron, A. Bicchi, "Conflict Resolution Problems for Air Traffic Management Systems Solved With Mixed Integer Programming", IEEE Trans. Intelligent Transport. Systems, Vol. 3, No. 1, March 2002

[PAR] S. Parsons, M. Wooldridge, "Game Theory and Decision Theory in Multi-Agent Systems", AAMAS, No. 5, 2002, Kluwer Academic Publishers

[TEL] Gerald Tel, "Introduction To Distributed Algorithms", 2nd ed., Cambridge Univ. Press, 2000

[TOS] P. Tosic, M. Jang, G. Agha, "Constraint Optimization Framework for UAV Surveillance Mission Problem", in preparation

[YOK] M. Yokoo, K. Hirayama, "Algorithms for Distributed Constraint Satisfaction: A review", AAMAS, Vol. 3, No. 2, 2000

Runtime Safety Analysis of Multithreaded Programs

Koushik Sen *

Dept. of Computer Science
University of Illinois at Urbana

ksen@uiuc.edu

Grigore Roșu †

Dept. of Computer Science
University of Illinois at Urbana

grosu@uiuc.edu

Gul Agha ‡

Dept. of Computer Science
University of Illinois at Urbana

agha@uiuc.edu

ABSTRACT

Foundational and scalable techniques for runtime safety analysis of multithreaded programs are explored in this paper. A technique based on vector clocks to extract the causal dependency order on state updates from a running multithreaded program is presented, together with algorithms to analyze a multithreaded computation against safety properties expressed using temporal logics. A prototype tool implementing our techniques, is also presented, together with examples where it can predict safety errors in multithreaded programs from successful executions of those programs. This tool is called Java MultiPathExplorer (JMPaX), and available for download on the web. To the best of our knowledge, JMPaX is the first tool of its kind.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification

General Terms

Verification

Keywords

LTL, predictive analysis, safety analysis, runtime monitoring, vector clock, multithreaded program, JMPaX, Java

*Supported in part by the Defense Advanced Research Projects Agency (Contract numbers: F30602-00-2-0586 and F33615-01-C-1907) and the ONR Grant N00014-02-1-0715.

†Supported in part by joint NSF/NASA grant CCR-0234524

‡Supported in part by the Defense Advanced Research Projects Agency (Contract numbers: F30602-00-2-0586 and F33615-01-C-1907) and the ONR Grant N00014-02-1-0715.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.
Copyright 2003 ACM 1-58113-743-5/03/0009 ...\$5.00.

1. INTRODUCTION

The purpose of this paper is to investigate foundational, scalable techniques for runtime safety analysis of multithreaded programs, i.e., programs in which several execution threads communicate with each other via shared variables and synchronization points, as well as to introduce a prototype tool, called Java MultiPathExplorer (JMPaX – see Figure 1), based on our foundational techniques, which can reveal errors in multithreaded programs that are hard or impossible to detect otherwise. The user of JMPaX specifies safety properties of interest, using a past time temporal logic, regarding the global state of a multithreaded program, which is already assumed in compiled form, calls an instrumentation script which automatically instruments the executable multithreaded program to emit relevant state update events to an external observer, and finally runs the program on any JVM and analyzes the safety violation messages reported by the observer. A particularly appealing aspect of our approach is that, despite the fact that a single execution, or interleaving, of a multithreaded program can be observed, a comprehensive analysis of *all possible executions* is performed; a possible execution is any execution which does not violate the observed causal dependency partial order on state update events. Thus, tools built on our techniques, including JMPaX, have the ability to *predict* safety violation errors in multithreaded programs by observing successful executions.

The work in this paper falls under the area recently called *runtime verification* [11, 10], a major goal of which is to combine testing and formal methods techniques. Testing scales well, and is by far the most used technique in practice to validate software systems. By merging testing and temporal logic specification, we aim to achieve the benefits of both approaches, while avoiding some of the pitfalls of ad hoc testing and the complexity of full-blown theorem proving and model checking. Of course, there is a price to be paid in order to obtain a scalable technique: the loss of coverage. The suggested framework can only be used to examine single execution traces, and therefore can not be used to *prove* a system correct. However, a single execution trace typically contains much more information than what appears at first sight. In this paper, we show how one can analyze all the other multithreaded executions that are hidden behind a particular observed execution. Our work is based on the belief that software engineers are willing to trade coverage for scalability, so our goals are to provide tools that use formal methods techniques in a lightweight manner, use unmodified programming languages or under-

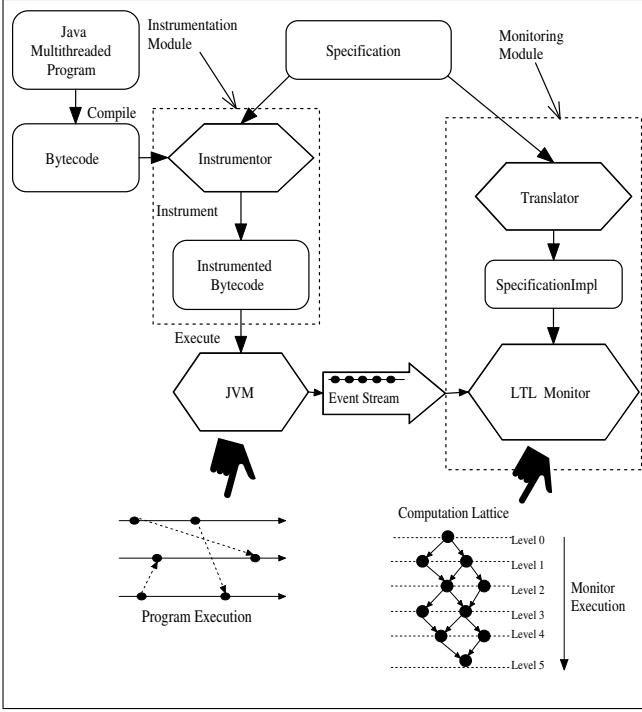


Figure 1: JMPaX Architecture

lying executional engines (such as JVMs), are completely automatic, implement very efficient algorithms and eventually find *many* errors in programs. A longer term goal is to explore the use of conformance with a formal specification to achieve error recovery. The idea is that a predicted failure may trigger an error-avoidance or recovery action in the monitored program.

The closest works in spirit to ours are NASA’s PathExplorer (PaX) and its Java instance JPaX [10, 9], which is a runtime verification system developed at NASA Ames, and UPENN’s MaC and its instance Java MaC [14, 15]. It is actually the latter’s limitations that motivated us to pursue our current research. The major limitation of these systems with regards to safety analysis is that they only analyze the observed run. Therefore, they can only detect existing errors in current executions; they do not have the ability to predict possible errors from successful runs. To be more precise in our claim, let us consider a real-life example where JMPaX was able to detect a violation of a safety property from a single execution of the program. However, the likelihood of detecting this bug only by monitoring the observed run, as JPaX and Java-MaC do, is very low. The example consists of a two threaded program to control the landing of an airplane. It has three variables `landing`, `approved`, and `radio`; their values are 1 when the *plane is landing*, *landing has been approved*, and *radio signal is live*, and 0 otherwise. The safety property that we want to verify is “If the plane has started landing, then it is the case that landing has been approved and since the approval the radio signal has never been down.” As shown in Subsection 3.1, this property can be formally written in our extension of past time linear temporal logic as the formula

$$\uparrow \text{landing} \rightarrow [\text{approved}, \downarrow \text{radio}]_s.$$

The code snippet for a naive implementation of this control program is given as follows:

```
int landing = 0, approved = 0, radio = 1;
void thread1(){
    askLandingApproval();
    if(approved==1){
        print("Landing approved");
        landing = 1;
        print("Landing started");
    } else {
        print("Landing not approved");
    }
}

void askLandingApproval(){
    if(radio==0) approved = 0;
    else approved = 1;
}

void thread2(){
    while(radio){checkRadio();}
}

void checkRadio(){
    randomly change value of radio;
}
```

The above code uses some dummy functions, namely `askLandingApproval` and `checkRadio`, which can be implemented in their entirety in a real scenario. The program has a serious problem which cannot be detected easily from a single run. The problem is as follows. Suppose the plane has received approval for landing and just before it started landing the radio signal went off. In this situation, the plane must abort landing. But this situation will very rarely arise in an execution: namely, when `radio` is set to 0 between the approval of landing and the start of actual landing. So a simple observer will not probably detect the bug. However, note that even if the radio goes off after the landing has started, JMPaX can still construct a possible run in which radio goes off between landing and approval. Thus JMPaX will be able to predict the safety violation from a single successful execution of the program. This example shows the power of our runtime verification technique as compared to JPaX and Java-MaC.

Other related approaches include model checking [6], especially Java bytecode model checking [8], and debugging of distributed systems. It is important to observe that, unlike model checking where all possible code interleavings are analyzed, in our approach to runtime safety analysis one actually runs the program and extracts causal dependencies among updates of the multithreaded program state, and then analyzes all possible interleavings that do not violate the causal dependency. At the expense of a lower coverage, our approach analyzes a significantly lower amount of thread interleavings than a typical model checker would normally do, so it scales up better. The safety properties that we analyze are more general than the simpler state predicates that are typically considered in the literature on debugging distributed systems (see for example [19, 4, 3]). We allow any past time linear temporal logic formula built on state predicates, so our safety properties can refer to the entire past history of states. An important practical aspect of our algorithm is that, despite the fact that there can be a potentially exponential number of runs (in the length of the runs), they can all be analyzed *in parallel*, by generating and traversing the computation lattice extracted from the

observed multithreaded execution on a level-by-level basis. The relevant information regarding the previous levels can be encoded compactly, so those levels do not need to be stored, thus allowing the memory to be reused.

We can think of at least three major contributions of the work presented in this paper. First, we nontrivially extend the runtime safety analysis capabilities of systems like JPaX and Java Mac, by providing the ability to predict safety errors from successful executions; we are not aware of any other efforts in this direction. Second, we underlie the foundations of relevant causality in multithreaded systems with shared variables and synchronization points, which one can use to instrument multithreaded programs to emit to external observers a causal dependency partial relation on global state updates via relevant events timestamped with appropriate vector clocks; this is done in Section 2, where, due to its foundational aspect, all the proofs of the claimed results are provided. Finally, a modular implementation of a prototype runtime analysis system, JMPaX, is given, showing that, despite their theoretical flavor, all the concepts presented in the paper are in fact quite practical and can lead to new scalable verification tools.

2. RELEVANT CAUSALITY IN MULTITHREADED SYSTEMS

We consider multithreaded systems in which several threads communicate with each other via a set of shared variables. The theme of this paper is to show how such a system can be analyzed for safety by an external observer that obtains relevant information about the system from messages sent by the system after appropriate instrumentation. The safety formulae refer to sets of shared variables, so these messages contain update information about those variables. A crucial observation here is that some variable updates can causally depend on others. For example, if a thread writes a variable x and then another thread writes y due to a statement $y = x + 1$, then the update of y causally depends upon the update of x . In this section we present an algorithm which, given an executing multithreaded system, generates appropriate messages to be sent to an external observer. The observer, in order to perform its more elaborated safety analysis, extracts the state update information from such messages together with the causality partial order among the updates.

Formally, given n threads p_1, p_2, \dots, p_n , a *multithreaded execution* is abstracted as a sequence of events $e_1 e_2 \dots e$, each belonging to one of the n threads and being of type either *internal* or *read* or *write* of a shared variable. We use e_i^j to represent the j -th event generated by thread p_i since the start of its execution. From now on in this section we assume an arbitrary but fixed multithreaded execution. When the process or the position of an event is not important then we can refer to the event generically, such as e, e' , etc.; we may write $e \in p_i$ when event e is generated by thread p_i . Let S be the set of shared variables. There is an immediate notion of *variable access precedence* for each shared variable $x \in S$: we say that e x -*precedes* e' , written $e <_x e'$, if and only if e and e' are variable access events (reads or writes) to the same variable x , and e “happens before” e' ; this “happens-before” relation can be easily realized by keeping a counter for each shared variable which is increased by each access to it. Let \mathcal{E} be the set of all the events of a multithreaded

execution, and let \prec be the partial order on \mathcal{E} defined as follows:

- $e_i^k \prec e_i^l$ if $k < l$;
- $e \prec e'$ if there is some $x \in S$ such that $e <_x e'$ and atleast one of e, e' is a write.
- $e \prec e''$ if $e \prec e'$ and $e' \prec e''$.

We write $e || e'$ when it is not the case that $e \prec e'$ or $e' \prec e$. A partial order on events \prec defined above is called a *multithreaded computation* associated with the original multithreaded execution. As shown in Subsection 3.3, synchronization can be treated very elegantly by generating appropriate read/write events, so that the notion of multithreaded computation as defined above is as general as currently needed. Note that the original multithreaded execution was used only to provide a total ordering on the read/write accesses of each shared variable.¹ A permutation of all the events e_1, e_2, \dots, e_r which does not violate the multithreaded computation is called a *consistent multithreaded run*, or simply, a *multithreaded run*.

Intuitively, $e \prec e'$, read as e' *causally depends upon* e , if and only if e occurred before e' in the given multithreaded execution and a change of their order does not generate a consistent multithreaded run. We argue that the notion of multithreaded computation defined above is the *weakest assumption* that an omniscient observer of the multithreaded execution can make about the program. Intuitively, this is because an external observer *cannot disregard* the order in which the same variable is modified and used within the observed execution, because this order can be part of the intrinsic logic of the multithreaded program. However, multiple consecutive reads of the same variable can be permuted, and the particular order observed in the given execution is not critical; it can be, for example, a result of a particular thread scheduling algorithm. By allowing an observer to analyze *multithreaded computations* rather than just *multithreaded run*, one gets the benefit of not only properly dealing with potential reordering of delivered messages (for example, due to using multiple channels in order to reduce the monitoring overhead), but also of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling.

Not all the variable in S are needed to evaluate the safety formula to be checked. To minimize number of messages sent to an observer, and for technical reasons discussed later, we consider a subset $\mathcal{R} \subseteq \mathcal{E}$ of *relevant events*. Then we define the \mathcal{R} -*relevant causality* on \mathcal{E} as the relation $\triangleleft := \prec \cap (\mathcal{R} \times \mathcal{R})$, so that $e \triangleleft e'$ if and only if $e, e' \in \mathcal{R}$ and $e \prec e'$. We provide a technique based on *vector clocks* [7, 17] that correctly implements the relevant causality relation.

Let V_i be an n -dimensional vector of natural numbers for thread p_i , for each $1 \leq i \leq n$, and let V_x^a and V_x^w be two additional n -dimensional vectors for each shared variable x ; we call the former *access vector clock* and the latter *write vector clock*. All the vector clocks are initialized to 0 at the beginning of computation. For two n -dimensional vectors we say that $V \leq V'$ if and only if $V[j] \leq V'[j]$ for all $1 \leq j \leq n$, and we say that $V < V'$ iff $V \leq V'$ and there is some $1 \leq j \leq n$ such that $V[j] < V'[j]$; also, $\max\{V, V'\}$ is

¹One could have defined a multithreaded computation more abstractly but less intuitively, by starting with a total order $<_x$ on the subset of events accessing each shared variable x .

the vector with $\max\{V, V'\}[j] = \max\{V[j], V'[j]\}$ for each $1 \leq j \leq n$. Whenever a thread p_i with current vector clock V_i processes event e_i^k , the following vector clock algorithm is executed:

1. if e_i^k is relevant, i.e., if $e_i^k \in \mathcal{R}$, then

$$V_i[i] \leftarrow V_i[i] + 1$$
2. if e_i^k is a read of a variable x then

$$V_i \leftarrow \max\{V_i, V_x^w\}$$

$$V_x^a \leftarrow \max\{V_x^a, V_i\}$$
3. if e_i^k is a write of a variable x then

$$V_x^w \leftarrow V_x^a \leftarrow V_i \leftarrow \max\{V_x^a, V_i\}$$
4. if e_i^k is relevant then
send message $\langle e_i^k, i, V_i \rangle$ to observer.

Then the following crucial results hold:

LEMMA 1. After event e_i^k is processed by thread p_i ,

- a. $V_i[j]$ equals the number of relevant events of p_j that causally precede e_i^k ; if $j = i$ and e_i^k is relevant then this number also includes e_i^k ;
- b. $V_x^a[j]$ equals the number of relevant events of p_j that causally precede the most recent event that accessed (read or wrote) x ; if $i = j$ and e_i^k is a relevant read or write of x event then this number also includes e_i^k ;
- c. $V_x^w[j]$ equals the number of relevant events of p_j that causally precede the most recent write event of x ; if $i = j$ and e_i^k is a relevant write of x then this number also includes e_i^k .

THEOREM 2. If $\langle e, i, V \rangle$ and $\langle e', j, V' \rangle$ are two messages sent by our algorithm, then $e \triangleleft e'$ if and only if $V[i] \leq V'[i]$. If i and j are not given, then $e \triangleleft e'$ if and only if $V < V'$.

In a summary, the above theorem states that the vector clock algorithm correctly implements causality in multithreaded programs. The detailed proofs of the above results are given in [18].

Consider what happens at the observer's site. The observer receives messages of the form $\langle e, i, V \rangle$ in any possible order. We let \mathcal{R} denote the set of received relevant events, which we simply call *events* in what follows. By using Theorem 2, the observer can infer the causal dependency between the relevant events emitted by the multithreaded system. Inspired by similar definitions at the multithreaded system's [2], we define the important notions of relevant multithreaded computation and run as follows. A *relevant multithreaded computation*, simply called *multithreaded computation* from now on, is the partial order on events that the observer can infer, which is nothing but the relation \triangleleft . A *relevant multithreaded run*, also simply called *multithreaded run* from now on, is any permutation of the received events which *does not violate* the multithreaded computation. Our purpose in this paper is to check safety requirements against *all* (relevant) multithreaded runs of a multithreaded system.

We assume that the relevant events are only writes of shared variables that appear in the safety formulae to be monitored, and that these events contain a pair of the name of the corresponding variable and the value which was written to it. We call these variables *relevant variables*. Note that events can change the state of the multithreaded system as seen by the observer; this is formalized next. A *relevant program state*, or simply a *program state* is a map

from relevant variables to concrete values. Any permutation of events generates a sequence of program states in the obvious way, however, not all permutations of events are valid multithreaded runs. A program state is called *consistent* if and only if there is a multithreaded run containing that state in its sequence of generated program states. We next formalize these concepts.

For a given permutation of (relevant) events in \mathcal{R} , say $e_1 e_2 \dots e_{|\mathcal{R}|}$, we can let e_i^k denote the k -th event of thread p_i for each $1 \leq i \leq n$. Then the relevant program state after the events $e_1^{k_1}, e_2^{k_2}, \dots, e_n^{k_n}$ is called a *relevant global multithreaded state*, or simply a *relevant global state* or even just *state*, and is denoted by $\Sigma^{k_1 k_2 \dots k_n}$. A state $\Sigma^{k_1 k_2 \dots k_n}$ is called *consistent* if and only if for any $1 \leq i \leq n$ and any $l_i \leq k_i$, it is the case that $l_j \leq k_j$ for any $1 \leq j \leq n$ and any l_j such that $e_j^{l_j} \triangleleft e_i^{l_i}$. Let $\Sigma^{00\dots 0}$ be the *initial global state*, $\Sigma^{00\dots 0}$. An important observation is that $e_1 e_2 \dots e_{|\mathcal{R}|}$ is a multithreaded run if and only if it generates a sequence of global states $\Sigma^{K_0} \Sigma^{K_1} \dots \Sigma^{K_{|\mathcal{R}|}}$ such that each Σ^{K_r} is consistent and for any two consecutive Σ^{K_r} and $\Sigma^{K_{r+1}}$, K_r and K_{r+1} differ in exactly one index, say i , where the i -th element in K_{r+1} is larger by 1 than the i -th element in K_r . For that reason, we will identify the sequences of states $\Sigma^{K_0} \Sigma^{K_1} \dots \Sigma^{K_{|\mathcal{R}|}}$ as above with multithreaded runs, and simply call them *runs*. We say that Σ *leads-to* Σ' , written $\Sigma \rightsquigarrow \Sigma'$, when there is some run in which Σ and Σ' are consecutive states. The set of all consistent global states together with the relation \rightsquigarrow forms a *lattice*. The lattice has n mutually orthogonal axis representing each thread. For a state $\Sigma^{k_1 k_2 \dots k_n}$, we call $k_1 + k_2 + \dots + k_n$ its *level*. A *path* in the lattice is a sequence of consistent global states on increasing level, where the level increases by 1 between any two consecutive states in the path. Therefore, a run is just a path starting with $\Sigma^{00\dots 0}$ and ending with $\Sigma^{r_1 r_2 \dots r_n}$, where r_i is the total number of events of thread i for each $1 \leq i \leq n$. Therefore, a multithreaded computation can be seen as a lattice; we call this lattice a *computation lattice*.

EXAMPLE 1. Suppose that one wants to monitor some safety property of the multithreaded program below. The program involves relevant variables x , y and z :

```
Initially: x = -1; y = 0; z = 0;
thread T1{ ... x++; ... y = x + 1; ... }
thread T2{ ... z = x + 1; ... x++; ... }
```

The ellipses (...) indicate code that is not relevant, i.e., that does not access the variables x , y and z . This multithreaded program, after appropriate instrumentation, sends messages to an observer whenever the relevant variables are updated. A possible execution of the program to be sent to the observer, described in terms of relevant variable updates, can be

$$\{x = -1, y = 0, z = 0\}, \{x = 0\}, \{z = 1\}, \{y = 1\}, \{x = 1\}$$

The first message to observer sends the initial state of the whole system as a set of variable-value pairs. The second event is generated when the value of x is incremented by the first thread. The above execution corresponds to the

sequence of program states

$$(-1, 0, 0), (0, 0, 0), (0, 0, 1), (0, 1, 1), (1, 1, 1)$$

where the tuple $(-1, 0, 0)$ denotes the state in which $x = -1, y = 0, z = 0$. Following the vector clock algorithm, we can deduce that the observer will receive the multithreaded computation in Figure 2 which generates the computation lattice shown in the same figure.

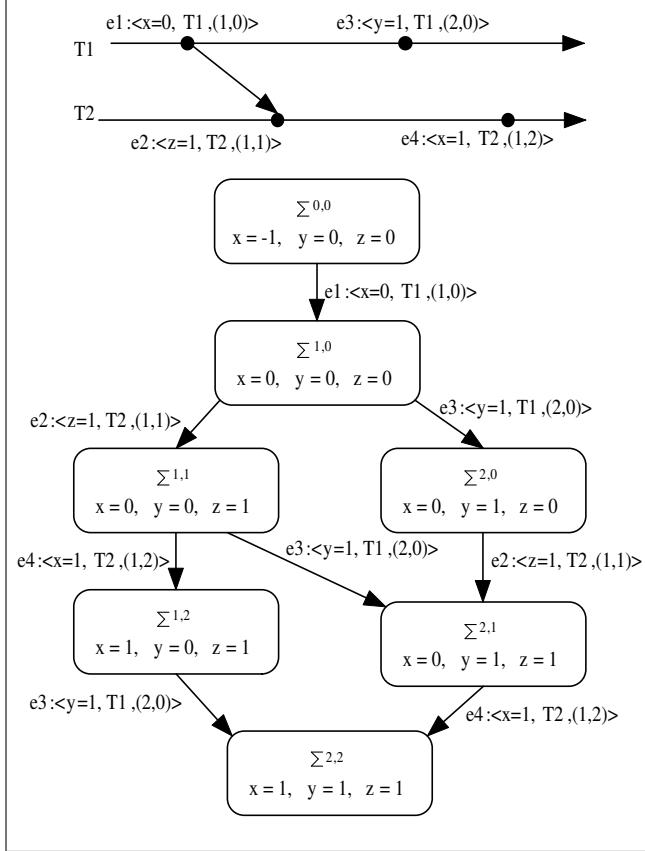


Figure 2: Computation lattice and three runs.

Notice that the observed multithreaded execution corresponds to just one particular multithreaded run out of the three possible. We will show that it is often possible that the observed run does not violate any safety property, but the run nevertheless shows that there are other possible runs that are not safe. We will propose an algorithm that will detect safety violations in any possible run, even though the violation was not revealed by the particular observed run. An appealing aspect of our algorithm is that, despite the fact that there can be a potentially exponential number of runs (in the maximum width of a level), they can all be analyzed *in parallel*, by generating and traversing the lattice on a level-by-level basis; the previous levels are not needed, so memory can be reused.

3. MULTITHREADED SAFETY ANALYSIS

In this section, we first introduce the past time temporal logic that we use to express safety properties, then we recall an algorithm to monitor such properties efficiently against

a single run, and finally we show how this algorithm non-trivially extends to monitoring multithreaded computations given as partial orders.

3.1 Safety in Temporal Logics

We use past time Linear Temporal Logic (*ptLTL*) [16] to express our safety properties. Our choice of past time linear temporal logic is motivated by two considerations:

1. It is powerful enough to express safety properties of concurrent systems;
2. The monitors for a safety formula written in *ptLTL* are very efficient; they perform linearly in the size of the formula in the worst case [12].

Now we briefly introduce the basic notions of *ptLTL*, and describe some new operators that are particularly useful for runtime monitoring. The syntax of *ptLTL* is given as follows:

$$\begin{aligned} F ::= & \text{ true } | \text{ false } | a \in A | \neg F | F \text{ op } F & \text{Propositional ops} \\ & \odot F | \Diamond F | \Box F | FS_s F | FS_w F & \text{Standard operators} \\ & \uparrow F | \downarrow F | [F, F)_s | [F, F)_w & \text{Monitoring ops} \end{aligned}$$

where *op* are the standard binary operators, namely \wedge , \vee , \rightarrow , \leftrightarrow , and $\odot F$ should be read as “previously”, $\Diamond F$ as “eventually in the past”, $\Box F$ as “always in the past”, $F_1 S_s F_2$ as “ F_1 strong since F_2 ”, $F_1 S_w F_2$ as “ F_1 weak since F_2 ”, $\uparrow F$ as “start F ”, $\downarrow F$ as “end F ”, $[F_1, F_2)_s$ as “strong interval F_1, F_2 ”, and $[F_1, F_2)_w$ as “weak interval F_1, F_2 ”.

The logic is interpreted on a finite sequence of states or a run. If $\rho = s_1 s_2 \dots s_n$ is a run then we let ρ_i denote the prefix run $s_1 s_2 \dots s_i$ for each $1 \leq i \leq n$. The semantics of the different operators is given in Table 1.

The monitoring operators \uparrow , \downarrow , $[,]_s$, and $[,]_w$ were introduced in [12, 15]. These operators have been found to be very intuitive and useful in specifying properties for runtime monitoring. Informally, $\uparrow F$ is true if and only if F starts to be true in the current state, $\downarrow F$ is true if and only if F ends being true in the current state, and $[F_1, F_2)_s$ is true if and only if F_2 was never true since the last time F_1 was observed to be true, including the state when F_1 was true; the interval operator has a strong and a weak version. For example, if *boot* and *down* are predicates on the state of a web server to be monitored, say for the last 24 hours, then $[boot, down)_s$ is a property stating that the server was rebooted recently and the since then it was not down, while $[boot, down)_w$ say that server was not unexpectedly down recently, meaning that it was either not down at all recently or it was rebooted recently and since then it was not down.

In runtime monitoring, we start the process of monitoring from the point the first event is generated and it continues as long as the events are generated. Thus given a *ptLTL* formula F we check whether $\rho_i \models F$ for all $1 \leq i \leq n$, where $\rho = s_1 s_2 \dots s_n$ is a finite run constructed from the events.

3.2 Checking Safety Against a Single Run

We describe an algorithm for monitoring the multithreaded execution or *the observed run* of a multithreaded computation, which is just one path in the computation lattice, and illustrate it through an example. This algorithm is a modified version of the algorithm presented in [12]. The algorithm computes the boolean value of the formula to be monitored, by recursively evaluating the boolean value of

$\rho \models \text{true}$	is true for all ρ ,
$\rho \models a$	iff a holds in the state s_n ,
$\rho \models \neg F$	iff $\rho \not\models F$,
$\rho \models F_1 \text{ op } F_2$	iff $\rho \models F_1$ and/or/implies/iff $\rho \models F_2$, when op is $\wedge / \vee / \rightarrow / \leftrightarrow$,
$\rho \models \odot F$	iff $\rho' \models F$, where $\rho' = \rho_{n-1}$ if $n > 1$ and $\rho' = \rho$ if $n = 1$,
$\rho \models \Diamond F$	iff $\rho_i \models F$ for some $1 \leq i \leq n$,
$\rho \models \Box F$	iff $\rho_i \models F$ for all $1 \leq i \leq n$,
$\rho \models F_1 \mathcal{S}_s F_2$	iff $\rho_j \models F_2$ for some $1 \leq j \leq n$ and $\rho_i \models F_1$ for all $1 \leq i \leq n$,
$\rho \models F_1 \mathcal{S}_w F_2$	iff $\rho \models F_1 \mathcal{S}_s F_2$ or $\rho \models \Box F_1$,
$\rho \models \uparrow F$	iff $\rho \models F$ and it is not the case that $\rho \models \odot F$,
$\rho \models \downarrow F$	iff $\rho \models \odot F$ and it is not the case that $\rho \models F$,
$\rho \models [F_1, F_2]_s$	iff $\rho_j \models F_1$ for some $1 \leq j \leq n$ and $\rho_i \not\models F_2$ for all $j \leq i \leq n$,
$\rho \models [F_1, F_2]_w$	iff $[F_1, F_2]_s$ or $\rho \models \Box \neg F_2$,

Table 1: Semantics of *ptLTL*

each of its subformulae in the current state. In the process it also uses the boolean value of certain subformulae evaluated in the previous state. Next we define this recursive function *eval*. The recursive nature of the temporal operators in *ptLTL* enables us to define the boolean value of a formula in the current state in terms of its boolean value in the previous state and the boolean value of its subformulae in the current state. For example we can define:

$$\begin{aligned}
\rho \models \Diamond F &\quad \text{iff } \rho \models F \text{ or } (n > 1 \text{ and } \rho_{n-1} \models \Diamond F) \\
\rho \models \Box F &\quad \text{iff } \rho \models F \text{ and } (n > 1 \text{ implies } \rho_{n-1} \models \Box F) \\
\rho \models F_1 \mathcal{S}_s F_2 &\quad \text{iff } \rho \models F_2 \text{ or } \\
&\quad (n > 1 \text{ and } \rho \models F_1 \text{ and } \rho_{n-1} \models F_1 \mathcal{S}_s F_2) \\
\rho \models F_1 \mathcal{S}_w F_2 &\quad \text{iff } \rho \models F_2 \text{ or } \\
&\quad (\rho \models F_1 \text{ and } (n > 1 \text{ and } \rho_{n-1} \models F_1 \mathcal{S}_w F_2)) \\
\rho \models [F_1, F_2]_s &\quad \text{iff } \rho \not\models F_2 \text{ and } \\
&\quad (\rho \models F_1 \text{ or } (n > 1 \text{ and } \rho_{n-1} \models [F_1, F_2]_s)) \\
\rho \models [F_1, F_2]_w &\quad \text{iff } \rho \not\models F_2 \text{ and } \\
&\quad (\rho \models F_1 \text{ or } (n > 1 \text{ implies } \rho_{n-1} \models [F_1, F_2]_w))
\end{aligned}$$

These definitions correspond to the code for the cases of the operators \Diamond , \Box , \mathcal{S}_s , \mathcal{S}_w , $[,]_s$, and $[,]_w$ in the function *eval*. The function *op(f)* returns the operator of the formula *f*, *binary(op(f))* returns *true* if *op(f)* is binary, *unary(op(f))* returns *true* if *op(f)* is true, *left(f)* returns the left subformula of *f*, *right(f)* returns the right subformula of *f*, and *subformula(f)* returns the subformula of *f*.

```

boolean
eval(Formula f, State s, array now, array pre, int index){
    if binary(op(f)) then{
        lval ← eval(left(f), now, pre, index);
        rval ← eval(right(f), now, pre, index);}
    else if unary(op(f)) then
        val ← eval(subformula(f), now, pre, index);
    case(op(f)) of{
        p : return p(s); true : return true; false : return false;
        op : return rval op lval; ¬ : return not val;
        Ss, Sw : now[+index] ← lval or rval;
        return (pre[index] and lval) or rval;
        [,]s, [,]w :
            now[+index] ← (not rval) and (pre[index] or lval);
            return (not rval) and (pre[index] or lval);
        ↑ : now[+index] ← val;
        return (not pre[index]) and val;
        ↓ : now[+index] ← val;
        return pre[index] and (not val);
    }
}

```

```

    ⊓ : now[+index] ← val; return pre[index] and val;
    ⊔ : now[+index] ← val; return pre[index] or val;
    ⊕ : now[+index] ← val; return pre[index];
}
}

```

Here, the *pre* array passed as an argument contains the boolean values of all subformulae in the previous state, that will be required in the current state. While the *now* array, after the evaluation of *eval* function, will contain the boolean values of all subformulae in the current state that will be required in the next state. Note, here the *now* array is passed as reference, and its value is set in the function *eval*. The function *eval*, however, cannot be used to evaluate the boolean value of a formula for the first state in a run, as the recursion handles the case $n = 1$ in a different way. We define the function *init* to handle this special case as follows:

```

boolean init(Formula f, State s, array now, int index){
    if binary(op(f)) then{
        lval ← init(left(f), now, index);
        rval ← eval(right(f), now, index);}
    else if unary(op(f)) then
        val ← init(subformula(f), now, index);
    case(op(f)) of{
        p : return p(s); true : return true; false : return false;
        op : return rval op lval; ¬ : return not val;
        Ss : now[+index] ← rval; return rval;
        Sw : now[+index] ← lval or rval; return lval or rval;
        [,]s : now[+index] ← (not rval) and lval;
        return (not rval) and lval;
        [,]w : now[+index] ← (not rval); return (not rval);
        ↑, ↓ : now[+index] ← val; return false;
        ⊓, ⊔, ⊕ : now[+index] ← val; return val;
    }
}

```

For a given formula *f*, we define the function *monitor*, that at each iteration, consumes an event in the run, generates the next state from that event, and evaluates the formula for the state generated:

```

monitor(Formula f, Run r = e1e2...en){
    State state ← {};
    array now, pre;
    state ← update(state, e1);
    val ← init(f, state, now, 0);
}

```

```

if (not val) then output('property violated');
for i = 2 to n do{
    pre  $\leftarrow$  now;
    state  $\leftarrow$  update(state, ei);
    val  $\leftarrow$  eval(f, state, now, pre, 0);
    if (not val) then output('property violated');
}
}

```

In the initialization phase, the *state* variable is created from the event *e₁*. The *now* array is then calculated by calling the function *init* on the current *state*. After the calculation the result of *init* is checked for falsity, and an error message is issued if the result is false. Otherwise, the main loop is started. The main loop goes through the run, starting from the second event. At each iteration, *now* is copied to *pre*, the current state is generated by consuming an event from the run, the formula *f* is evaluated in the current state using the function *eval*, the result of evaluation is tested for falsity and an error message is generated if the result is false.

The time complexity of this algorithm is $\Theta(mn)$, where *m* is the size of the original formula and *n* is the length of the run. However, memory required by the algorithm² is $2m'$, *m'* being the number of temporal and monitor operators in the formula.

We now go back to the EXAMPLE 1. Suppose that one want to monitor the safety property $(x > 0) \rightarrow [(x = 0), y > z]_s$ on that program. The formula states that "if $(x > 0)$, then $(x = 0)$ has been true in the past, and since then $(y > z)$ was always false."

For the possible execution or the observed run of the program mentioned in Section 2, we have the following sequence of global states,

$(-1, 0, 0), (0, 0, 0), (0, 0, 1), (0, 1, 1), (1, 1, 1)$

where the tuple $(-1, 0, 0)$ denotes the state in which $x = -1, y = 0, z = 0$. The formula is satisfied in all the states of the sequence and so we say that the property specified by the formula is not violated by the given run. However, another possible run of the same computation is,

$\{x = -1, y = 0, z = 0\}, \{x = 0\}, \{y = 1\}, \{z = 1\}, \{x = 1\}$

This run corresponds to the sequence of states

$(-1, 0, 0), (0, 0, 0), (0, 1, 0), (0, 1, 1), (1, 1, 1)$

The formula is clearly violated in the last state of this sequence. This is because, $x > 0$ in the 5th state. This means that from 2nd state, in which $x = 0$, up to 5th state $y > z$ must be false. However, $y > z$ in the 3rd state. This violates the formula. Therefore, the monitoring algorithm that considers only the observed run presented in this subsection fails to detect this violation. In the next subsection we propose an algorithm that will detect such a potential bug from the original successful run.

3.3 Checking Safety Against All Runs

The algorithm, presented in the previous subsection, can only monitor a single run. As noticed earlier, monitoring one

²Here we assume that the recursive version is properly converted into an iterative algorithm using cps transform.

run may not reveal a bug that might be present in other possible runs. Our algorithm removes this limitation by monitoring all the possible runs of a multithreaded computation. The major hurdle in monitoring all possible runs is that the number of possible runs can be exponential in the length of the computation. We avoid this problem in our algorithm by traversing the computation lattice level by level. The events are generated by the algorithm presented in Section 2. The monitoring module consumes these events one by one, and monitors the safety formula on the computation lattice constructed from the events. We now describe the monitoring module in more details.

The monitoring module maintains a queue of events. Whenever an event arrives from the running multithreaded program, it *enqueues* it in the event queue (*Q*). The module also maintains a set of global states (*CurrentLevel*), that are present in the current level of the lattice. For each event *e* in the event queue, it tries to construct a global state from the set of states in the current level and the event *e*. If the global state is created successfully it is added to the set of global states (*NextLevel*) for the next level of the lattice. Once a global state in the current level becomes *unnecessary*, it is removed from the set of global states in the current level. When the set of global states in the current level becomes empty, we say that the set of global states for the next level is *complete*. At that time the module checks the safety formula (by calling *monitorAll(NextLevel)*) for the set of states in the next level. If the formula is not violated it marks the set of global states for the next level as the set of states for the current level, removes unnecessary events from the event queue, and restarts the iteration. The pseudocode for the process is given below:

```

for each (e  $\in Q$ ){
    if  $\exists s \in CurrentLevel$  s.t. isNextState(s, e) then
        NextLevel  $\leftarrow addToSet(NextLevel, createState(s, e));
    if isUnnecessary(s) then remove(s, CurrentLevel);
    if isEmpty(CurrentLevel) then{
        monitorAll(NextLevel);
        CurrentLevel  $\leftarrow NextLevel; NextLevel  $\leftarrow \{\};
        Q  $\leftarrow removeUnnecessaryEvents(NextLevel, Q);
    }
}$$$$ 
```

Every global state *s* contains the value of all relevant shared variables in the program, a *n*-dimensional vector clock *VC(s)* to represent the latest events from each thread that resulted in that global state, and a vector of boolean values called *flags*. Each component of *flags* is initially set to *false*. Here the predicate *isNextState(s,e)*, checks if the event *e* can convert the state *s* to a state *s'* in the next level of the lattice. The pseudocode for the predicate is given below:

```

boolean isNextState(s, e){
    i  $\leftarrow threadId(e);
    if VC(s)[i] = VC(e)[i] + 1 then{
        flags(s)[i] = true;
        if  $(\forall 1 \leq j \leq n, j \neq i) VC(s)[j] \geq VC(e)[j]$  then
            return true; else return false;
    }
    else return false;
}$ 
```

where *n* is the number of threads, *threadId(e)* returns the

index of the thread that generated the event e , $VC(s)$ returns the vector clock of global state s , $VC(e)$ returns the vector clock of event e , and $flags(s)$ returns the vector $flags$ associated with s . Note, here $flags(s)[i]$ is set to *true* if $VC(s)[i] = VC(e)[i] + 1$. This means that e is the only event from thread i that *can possibly* take state s to a state s' in the next level. When all the components of the vector $flags(s)$ become *true*, we say that the state s is *unnecessary*. Thus the function $isUnnecessary(s)$ checks if $(\forall 1 \leq i \leq n) flags(s)[i] = true$, where n is the number of threads.

The function $createState(s, e)$ creates a new global state s' , where s' is a possible global state that can result from s after the event e . For the purpose of monitoring we maintain, with every global state, a set of *pre* arrays called *PreSet*, and a set of *now* arrays called *NowSet*. In the function $createState$ we set the *PreSet* of s' with the *NowSet* of s . The pseudocode for $createState$ is as follows:

```
State createState( $s, e$ ){
     $s' \leftarrow \text{new copy of } s;$ 
     $j \leftarrow \text{threadId}(e); VC(s')[j] \leftarrow VC(s)[j] + 1;$ 
    for  $i = 1$  to  $n$  {flags( $s'$ )[ $i$ ]  $\leftarrow \text{false};$  }
    state( $s'$ )[var( $e$ )  $\leftarrow \text{value}(e)]; \text{return } s';$ 
    PreSet( $s')$   $\leftarrow \text{NowSet}(s);$ 
}
```

Here $state(s')$ returns the value of all relevant shared variables in state s' , $var(e)$ returns the name of the relevant variable that is written at the time of event e , $value(e)$ is the value that is written to $var(e)$, and $state(s')[var(e) \leftarrow value(e)]$ means that in $state(s')$, $var(e)$ is updated with $value(e)$.

The function $addToSet(NextLevel, s)$ adds the global state s to the set *NextLevel*. If s is already present in *NextLevel*, it updates the existing states' *PreSet* with the union of the existing states' *PreSet* and the *PreSet* of s . Two global states are same if their vector clocks are equal. The function $removeUnnecessaryEvents(CurrentLevel, Q)$ removes from Q the events that cannot contribute to the construction of any state at the next level. To do so, it creates a vector clock V_{min} whose each component is the minimum of the corresponding component of the vector clocks of all the global states in the set *CurrentLevel*. It then removes all the events in Q whose vector clocks are less than or equal to V_{min} . This function makes sure that we do not store the unnecessary events.

The function $monitorAll$ performs the actual monitoring of the formula. In this function, for each state s in the set *NextLevel*, we invoke the function *eval* (as discussed in the previous section) on s , for each *pre* array in the set *PreSet*. If *eval* returns *false*, we issue a ‘property violation’ warning. The *now* array that resulted from the invocation of *eval* is added to the set *NowSet* of s . The pseudocode for the function $monitorAll$ is given as follows:

```
monitorAll( $NextLevel$ ){
    for each  $s \in NextLevel$ {
        for each  $pre \in \text{PreSet}(s)$ {
            now  $\leftarrow \{\}; result \leftarrow \text{eval}(f, s, now, pre, 0);$ 
            if not result then output('property violated');
            NowSet( $s$ )  $\leftarrow \text{NowSet}(s) \cup \{now\};$ 
        }
    }
}
```

If the multithreaded program has synchronization blocks,

then we introduce, during instrumentation, a dummy shared variable that is read whenever we enter the synchronization block and is written when we exit the block. This makes sure that all the events in one execution of the block are causally dependent on the events in another execution of the same block, so that the interleaving between them becomes impossible.

Here the size of each *pre* array or *now* array is m' , where m' is the number of temporal operators in the formula. Therefore, the size of the set *PreSet* or the set *NowSet* can be atmost $2^{m'}$. This implies that the memory required for each state in the lattice is $O(2^{m'})$. If the maximum width of the lattice is w , then the total memory required by the program is $O(w2^{m'})$. The time taken by the algorithm at each iteration is $O(w2^m)$, where m is the size of the formula. However, if the threads in a program have very few dependency or synchronization points, then the number of valid permutations of the events can be very large, and therefore the width of the lattice can become large. To handle those situations we can add a parameter to the algorithm which specifies the maximum width of the lattice. Then, if the number of states in a level of the lattice becomes larger than the maximum width, the algorithm can be modified to consider only the most *probable states* in the level. We can specify different heuristics to calculate the most probable states in a given level of the lattice.

4. IMPLEMENTATION

We have implemented our monitoring algorithm, in a tool called Java Multi PathExplorer (JMPaX)[1], which has been designed to monitor multithreaded Java programs. The current implementation, see Figure 1, is written in Java and it assumes that all the shared variables of the multithreaded program are static variables of type *int*. The tool has two main modules, the *instrumentation* module and the *monitoring* module. The instrumentation program, named **instrument**, takes a specification file, a port number, and a list of class files as command line arguments. An example of such command is

```
java instrument spec server 7777 A.class B.class C.class
```

where the specification file **spec** contains a list of named formulae. The specification for the example discussed in Section 2 looks as follows:

```
F = (A.x > 0) -> [(A.x = 0), (A.y > A.z))_s
```

where **A** is the class containing the shared variables **x**, **y** and **z** as static fields. The program **instrument** instruments the classes, provided in the argument, as follows:

- i) It adds *access* and *write* vector clocks as static fields for each shared variable;
- ii) It adds code to create a vector clock whenever a thread is created;
- iii) For each read and write access of the shared variables in the class files, it adds codes to update the vector clocks according to the algorithm mentioned in Section 2;
- iv) It adds codes to send messages to the **server** at the port number 7777 for all writes of relevant variables.

To do so, the **instrument** program extracts the relevant variables from the specification file.

The instrumentation module uses BCEL [5] Java library to modify Java class files. We use the BCEL library to get a better handle for a Java classfile. It enables us to insert vector clocks as static member fields in a class, that is otherwise not possible with the tool JTrek [13]. We also make the update of vector clocks associated with a read or write, atomic through **synchronization**. For this we need to add Java bytecode both before and after the instructions **getstatic** and **putstatic**, that access the shared variables. This task is easier in BCEL as compared to JTrek.

A **translator**, which is part of monitoring module, reads the specification and generates a single Java class file, named **SpecificationImpl.class**. The monitoring module starts a server to listen events from the instrumented program, parses them, enqueues them in a queue, executes **translator** to generate **SpecificationImpl.class**, dynamically loads the class **SpecificationImpl.class**, and starts monitoring the formulae on the queue of events. It issues a warning whenever a formula is violated.

One of the test cases that we have implemented is the landing example described in Section 1. JMPaX was able to detect violation of a safety property from a single execution of the program. The safety property that we verified was:

$$\uparrow \text{landing} \rightarrow [\text{approved}, \downarrow \text{radio}]_s.$$

From a single execution of the code in which the radio went off after the landing, JMPaX constructed a possible run in which radio goes off between landing and approval, and hence it detected the safety violation. This example shows the power of our runtime verification technique.

5. CONCLUSION AND FUTURE WORK

We have investigated the problem of runtime analysis of multithreaded systems from a fundamental perspective. We have developed scalable techniques for extracting relevant events and their causal dependency from an executing multithreaded program. We have proposed and implemented algorithms to check safety properties against the computation lattice of a multithreaded computation. We have also briefly presented our prototype tool Java MultiPathExplorer, abbreviated JMPaX, which, at our knowledge, is the first tool that can predict violations of safety properties expressed in temporal logics from correct executions of multithreaded programs. We have also shown that, despite the fact that our safety properties can refer to any state in the past and that there is a potentially exponential number of multithreaded runs to be analyzed, one does not need to actually store the previous states; one can analyze all the multithreaded runs in parallel, by traversing the computation lattice top down, level-by-level.

Three major contributions have been made. First, we have nontrivially extended the capabilities of systems like JPaX and Java Mac, by providing the ability to *predict safety errors* from successful executions; we regard safety prediction as an important trade-off towards avoiding the inherent complexity of full-blown theorem proving and model checking; we are not aware of any other efforts in this direction. Second, we have defined the notion of relevant causality in multithreaded systems with shared variables

and synchronization points and we have provided a technique of implementing relevant causality based on vector clocks. Finally, we have implemented a modular prototype runtime analysis system, JMPaX; modularity comes from the fact that its instrumentation module can be used together with other computation lattice analysis tools, while its safety computation analysis module can be used in any event based setting, for example a distributed system. In fact, we intend to soon extend our work to analyzing arbitrary distributed systems at runtime for not only safety but also other properties of interest. There are also plans on developing a predictive analysis runtime environment for both multithreaded and distributed systems, as well as developing a GUI for JMPaX that would make it easy to use and understand by ordinary software engineers. Since our work is partly sponsored by NASA, we also intend to soon use JMPaX on real-world NASA-related large applications.

6. REFERENCES

- [1] Java Multi PathEXplorer, March 2003.
<http://fs1.cs.uiuc.edu/jmpax/>.
- [2] H. W. Cain and M. H. Lipasti. Verifying sequential consistency using vector clocks. In *Proceedings of the 14th annual ACM symposium on Parallel algorithms and Architectures*, pages 153–154. ACM, 2002.
- [3] C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.
- [4] R. Cooper and K. Marzullo. Consistent detection of global predicates. *ACM SIGPLAN Notices*, 26(12):167–174, 1991. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.
- [5] M. Dahm. Byte code engineering with the bcel api. Technical Report B-17-98, Freie Universit at Berlin, Institut für Informatik, April 2001.
- [6] J. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.
- [7] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed debugging*, pages 183–194. ACM, 1988.
- [8] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Apr. 2000.
- [9] K. Havelund and G. Rošu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
- [10] K. Havelund and G. Rosu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. IEEE, 2001.
- [11] K. Havelund and G. Rošu. *Runtime Verification 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001. Proceedings of a *Computer Aided Verification (CAV'01)* satellite workshop.
- [12] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Proceedings Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, pages 342–356, 2002.

- [13] JTrek Compaq Corp. www.digital.com/java/download/jtrek/.
- [14] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
- [15] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
- [16] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer-Verlag N.Y., Inc., 1995.
- [17] F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier science, 1989.
- [18] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. Technical Report UIUCDCS-R-2003-2334, University of Illinois at Urbana Champaign, April 2003.
- [19] S. D. Stoller. Detecting global predicates in distributed systems with clocks. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, pages 185–199, 1997.

*

Simple Genetic Algorithms for Pattern Learning: The Role of Crossovers

Predrag Totic*, **Gul Agha**
Open Systems Laboratory, Department of Computer Science
University of Illinois at Urbana-Champaign
{p-totic, agha}@cs.uiuc.edu

Abstract

Genetic algorithms (GA) are a biology-inspired class of evolutionary stochastic optimization techniques. The three basic operations in GA are selection, mutation and crossover. Of these three, we argue that only the crossover is a uniquely GA operation. We offer alternative interpretations of the basic GA operations, and then focus on how the speed of convergence changes with the change in crossover frequency in case of some simple pattern learning problems. Our position is that only those combinatorial optimization and search problems where crossovers significantly affect performance warrant an application of GA instead of simpler random search approaches.

Keywords: *genetic algorithms, stochastic optimization, pattern learning*

1 Brief Introduction to GA

Genetic algorithms (GA) [1, 2] are a class of stochastic optimization and search algorithms inspired by biology and, in particular, evolution and natural selection. *GA* have found numerous applications in a number of problem domains where a *randomized local search* of the parameter space is applicable. Some examples of such applications are various pattern detection and recognition problems [3] and training the weights of artificial neural networks [4].

GA belong to a class of *evolutionary algorithms*: the solution of a particular problem is sought via evolving a population of agents (represented as a set of strings of genes) from one generation to the next, until, after some number of generations, the desired quality of the population is reached. The goodness of individual organisms, as well as entire populations, is quantified by an appropriate *fitness function* over the gene strings.

The three basic operations in *GA*, in accordance with the original biological inspiration, are called *selection*, *mutation* and *crossover*.

Selection assures that, among the current pool of genes, those gene patterns whose *fitness* is comparably high are more likely to be kept in the next generation gene pool than the gene patterns of lower fitness. In the context of stochastic optimization, the “moves” or choices that have produced more desirable outcomes in the past are more likely to be chosen in the future than those moves or choices that have led to less favorable outcomes.

Mutation is a random, typically small perturbation that introduces some randomization into the search. It can be viewed as a version of “flipping a bit” (or, in this case, a gene) in gene sequences of some, randomly selected, “organisms” present in the current generation. While often detrimental in case of individual living organisms, mutations can ensure maintaining or even creating some *genetic diversity* in populations, as we discuss in §2. In particular, in the context of stochastic optimization, mutations help in exploring more of the search space than what would be typically explored with selection and crossovers only¹.

Crossover is an operation characteristic of the *sexual reproduction*: two different gene sequences are selected, and then appropriately chosen subsequences exchanged. Thus the two “children” sequences inherit some of their genes from each of the two parents; however, these children, assuming nontrivial subsequences have been crossed over, are also different from each of their parents as well as from one another. It has been argued that, in case of the natural evolution and many stochastic optimization problems alike, the interaction of these three basic operations and the synergy among them is what makes *GA* powerful in finding high fitness solutions, and that crossovers play a central role in this synergy.

In this paper, we mainly focus on the role of the crossover operation: how the convergence of a *GA* is affected as the *crossover rate* is varied, and, in particular, as it approaches zero. We argue that crossover is the only among the basic *GA* operations that is truly a specific feature of genetic algorithms, rather than a familiar operation from other random search and machine learning techniques, only “in disguise”. That non-negligible frequency of crossovers is quite essential for successful optimization via *GA* in some cases, while for all practical purposes crossovers are altogether unnecessary in other, is illustrated with some simple pattern learning examples. Some lessons drawn from our experiments (discussed in §3) are that (i) the choice of the actual crossover frequency, in case of those problems where crossovers are useful, is likely to be of utmost importance for the speed of convergence of the initial population to the one of maximal fitness, and that (ii) the good crossover and mutation frequencies (or frequency ranges) can be dependent on one another. The main lesson that we emphasize herein is that (iii) if the *GA* convergence rates are independent or nearly independent of the crossover rates, than the problem at hand is not a good *GA* application; in particular, we deduce from the examples in §3 that (iv) if a particular *GA* does quite well in a given context when the probability of crossovers is kept

*Contact author

¹Strictly, this need not always be the case - but often enough it is; the potential benefits of random mutations depend on the concrete problem at hand and its representation, the population size, and the “genetic diversity” of the initial (usually randomly chosen) population.

very small or even set to zero, then a *GA*-based approach is likely an “overkill” for that particular problem and/or application domain. Due to space constraints, in the discussion of our experiments in §3 we mostly focus on (iii)-(iv) and only briefly reflect on the other points above.

2 Interpreting GA Operations

We next briefly discuss the three basic *GA* operations. We interpret and characterize selection, mutation and crossover in terms of the basic concepts from *learning* and *optimization*. Our main point is that, while selection and mutation, in our view, can be readily understood without any reference to biology, natural evolution or genetic algorithms *per se*, the crossover operation is peculiarly a *GA* operation. Therefore, since any problem that can be solved by *GA* can also be solved by other randomized local search techniques, only those problems where a non-negligible probability of crossovers makes a considerable performance difference, truly deserve to be considered *GA* applications. A rigorous mathematical formulation of the three basic *GA* operations can be found, e.g., in *Chapter §2* of [3].

We fix some notation and terminology used throughout the paper. For concreteness, a “gene” is a symbol from either the ternary alphabet $\{A, B, C\}$, or its binary subset $\{A, B\}$. The “organisms” or agents are going to be represented as finite strings of genes over the alphabet. The evaluation function in *GA* is called *fitness*; in our case, it maps some finite set of binary or ternary strings into a bounded set of integer values. Average fitness of a generation of organisms is simply the arithmetic mean of all individual organisms’ fitness values. We shall use biology, mathematics and computer science terminologies interchangeably throughout the paper; for instance, the terms *organism*, *agent*, *individual* and *string (of genes or symbols)* are used synonymously herewith.

2.1 Selection

Selection is the *GA* way of achieving *reinforcement* or *exploitation* of the desirable properties that have already been discovered. In our simple model of selection, given a collection of strings and their individual fitness values, one or more low fitness strings are chosen to be discarded, and replaced by copies of the same number of high fitness strings. Thus, the size of the population remains the same, but its “genetic qualities” are improved via selection. Then genetic operations of mutation and crossover are performed on thereby improved population.

More generally, selection is a mechanism that ensures that the more fit individuals are to propagate their *genetic material* to the future generations with higher probability than the less fit individuals. Hence, selection can be viewed as a *reinforcement mechanism*: good gene patterns, in general, tend to be reinforced from one generation to the next, whereas low fitness gene patterns have a below-average probability of survival in each generation and, consequently, tend to disappear altogether in the long run [2, 5].

2.2 Mutation

There are at least two different roles of mutations, and hence two different ways of interpreting them. One is the small perturbation view: in order to prevent the algorithm from getting “stuck” (by, say, prematurely converging to a local

optimum), a random mutation breaks the current equilibrium until, eventually, a truly globally stable configuration is reached. Another role of mutations is that they enable the exploration of those parts of the configuration space that, in case of bad luck with the initial configuration, would have never been reached solely via the selection and crossover operations. A simple example is an optimization problem over ternary alphabet $\{A, B, C\}$ where the string encoding of any globally optimal solution includes the symbol C , yet the initial population of strings includes symbols A and B only. Clearly, without a nonzero probability of mutation of either A or B into C , no sequence of selection and crossover operations *alone* would ever be able to reach a globally optimal configuration.

Thus, while selection provides *exploitation* of the good traits in the present generation, mutations provide *randomization*, thereby expanding the search and enabling *exploration* of those configurations that otherwise may be inaccessible. In biological terms, while the selection operation, just as in case of *natural selection*, favors the survival, reproduction and ultimate numerical dominance of the fittest, the mutation operation enables creation and long-term persistence of *population diversity*.

2.3 Crossover: Uniquely GA

While selection is a model of reinforcement and mutation a form of random perturbation, explaining *crossover* in the terms familiar to an engineer or a computational scientist with no knowledge of biology is much more challenging. Biologically, crossovers are a peculiar feature of *sexual reproduction*. While a single crossover operation between two strings of symbols can in principle always be simulated via an appropriate sequence of single-symbol mutations, this does not lead to a natural interpretation of crossovers in terms of mutations. Namely, mutations are supposed to be infrequent, random, and independent of one another, rather than very common² and, in the case of ‘mutations’ simulating the same single crossover, highly mutually correlated and non-random³.

It has been argued that the power of *GA* stems primarily from the power of crossovers [3]. We rephrase this by arguing in §3 that those *GA* applications that use little or no crossovers, and/or perform independently of the crossover probabilities, are not genuine *GA*, in that the underlying problems can be readily and just as efficiently solved without genetic algorithms altogether.

3 Pattern Learning Examples

We now describe our experiments, outline the main features of the two pattern learning problems we have studied, present some of the experimental results, and then focus on their interpretation and implications.

In both problems, the (initial) string (or “chromosome”) length is eight. In the first set of our simple pattern learning experiments, the goal is to evolve a population of organisms or agents to the one where each agent, represented as a binary string over the alphabet $\{A, B\}$, contains in its representation each of these three patterns: *AAA*, *BBB*, and *BAB*. Given a string, its fitness is increased by 1 for the presence of each of these three substrings. The award is given irrespective of the number of repetitions;

²In practice, the crossover rates are usually by at least one order of magnitude higher than the mutation rates.

³Indeed, in case of binary alphabets, such mutations are uniquely determined once the two strings to be crossed over, and ‘the point of crossover’, have been chosen.

thus, fitness of *AAAAAAA* is only 1, and so is that of *BABBAABB*. String *BABAAAAAA* is of fitness 2, whereas string *BBBABAAAA* has fitness 3. In stochastic optimization terms, the fitness function reaches its global maximum for strings such as *BBBABAAA* above. In the reinforcement learning terms, the populations are iteratively trained to learn that the most desirable configurations are those such as *BBBABAAA*, good but suboptimal those such as *BBBAAAAAA*, while *BABABABA* is outright bad (zero fitness). We call this pattern learning task Problem 1.

In the second set of experiments, the optimal gene strings are those whose *longest all-A substrings* are of length four. That is, strings such as *AABBAAAAA* or *ABAAAAABA* are globally optimal (max. fitness 3), while, e.g., *BBAABABA* and *BAAAAABA*, with exactly three and five consecutive As in their *longest-all-A substrings*, respectively, are assigned fitness 2, the strings with exactly two or six consecutive As are assigned fitness 1, and all other strings are of fitness zero. We refer to this pattern learning problem as Problem 2.

Let us mention some of the main features of these two problems. It has been often stressed (e.g., §3 in [3]) that *GA* show their strength to the fullest extent in *multi-modal* optimization problems that, in addition to one or more global optima, also have several local optima. For such problems, it has been argued that *GA* distinguish themselves among other randomized search techniques for their ability to get out of locally optimal solutions and reliably eventually reach globally optimal configurations. For the string lengths of eight or greater, both of our problems allow several globally optimal solutions. An example of a locally optimal solution in case of our problems would be a string of fitness 2 whose fitness cannot be increased to 3 by flipping a single gene. In the case of Problem 1, string *BABBBBBB* is locally optimal in this sense. Problem 2, on the other hand, may or may not have such local maxima, depending on the string length. While $AAA(BAAA)^k$ is a strict local maximum for string lengths $4k+3$ (where $k = 1, 2, \dots$), there are no interesting local optima for other string lengths, including length 8 that we have used in most of our experiments⁴. Indeed, for length-eight strings, any relatively good but globally suboptimal string such as, e.g., *BBBAAABB*, can be made optimal by flipping a single symbol. That is, no such globally suboptimal string is nontrivially locally optimal for Problem 2, as there are always strings with higher fitness within Hamming distance 1 from it. Therefore, one may intuitively expect that *GA* would perform better in Problem 1 than in Problem 2. If “doing better” means that a typical genetic algorithm would, on average, converge faster in case of an instance of Problem 1 than in case of a comparable instance of Problem 2, this conjecture turns out to be wrong, as our experimental results indicate. A better measure of the applicability and usefulness of *GA*-based learning, in our view, is the variability of performance with respect to the *uniquely-GA* parameter, the crossover rate.

We used a simple, single-point crossover operator. We did allow this operator to be non-symmetric, in that the lengths of the offspring strings could differ from the lengths of their parents. In the case of Problem 2, this was utterly irrelevant - not surprisingly, given that crossovers themselves had very little, if any, impact there. However, this feature was absolutely crucial for allowing a reasonably rapid convergence in Problem 1 (see below).

Tables 1-2 pertain to the experiments for Problems 1 and 2, respectively, for small population sizes. We have tested

a broad range of crossover and mutation rate values; as our focus is on dependence of the speed of convergence on the crossover rate, we show some results for fixed mutation rate of 2%.

Table 1: Problem 1 (population: 10)

mutation rate: $p_m = 2\%$	# of agents: 10	# of experimental runs: 500
crossover rate, p_c	mean	standard deviation
0.90	290.90	277.32
0.80	128.76	129.61
0.70	66.70	56.51
0.60	43.50	31.63
0.50	33.48	21.03
0.45	32.66	22.96
0.40	33.34	21.06
0.35	30.45	22.48
0.30	31.36	23.28
0.25	29.53	21.61
0.20	31.12	24.10
0.15	34.53	30.24
0.10	39.39	31.23
0.05	57.55	52.46
0.02	80.11	83.88
0.01	102.46	110.90
0.001	192.11	257.69
0.00	207.34	302.80

Table 2: Problem 2 (population: 10)

mutation rate: $p_m = 2\%$	# of agents: 10	# of experimental runs: 500
crossover rate, p_c	mean	standard deviation
1.00	17.48	19.70
0.90	16.08	16.92
0.80	18.28	18.97
0.70	19.60	21.65
0.60	17.91	21.14
0.50	18.98	18.12
0.40	18.74	19.82
0.30	18.16	19.42
0.20	17.91	19.76
0.10	19.11	18.47
0.01	18.64	19.69
0.00	18.13	19.76

Tables 3-4 show the mean rates of convergence and the standard deviations when the population sizes are increased from 10 to 50. As the selection mechanism was kept the same (a single worst string in a given generation is overwritten by a copy of a single best string), the convergence is slower in case of bigger population sizes, as one would expect.

Table 3: Problem 1 (population: 50)

mutation rate: $p_m = 1\%$	# of agents: 50	# of experimental runs: 100
crossover rate, p_c	mean	standard deviation
0.60	225.59	358.55
0.50	94.48	52.27
0.45	77.58	29.08
0.40	73.13	19.49
0.35	67.88	18.50
0.30	67.17	16.38
0.25	63.83	13.71
0.20	65.42	17.19
0.15	64.45	17.97
0.10	66.38	24.39
0.05	72.72	33.57
0.02	93.53	62.62
0.01	106.73	90.92
0.001	218.17	314.74
0.00	306.36	626.58

⁴One can view a string such as *BBBBBBBB* as a trivial local maximum, in a sense that the fitness of such a string cannot be improved by flipping just one of the genes. However, we don't consider any such zero-fitness example an interesting locally optimal solution.

Table 4: Problem 2 (population: 50)

mutation rate: $p_m = 2\%$	# of agents: 50	# of experimental runs: 100
crossover rate, p_c	mean	standard deviation
1.00	48.67	3.63
0.90	49.09	4.96
0.80	49.25	3.61
0.70	48.84	3.67
0.60	48.43	3.32
0.50	48.63	3.05
0.40	48.53	3.47
0.30	49.00	3.19
0.20	49.32	4.92
0.10	48.79	3.51
0.00	49.44	3.24

In the case of Problem 1, as the crossover rate approaches zero, the convergence slows down considerably, as both *Table 1* and *Table 3* indicate. This dependence on the crossover rate p_c is highly nonlinear in two ranges: when p_c approaches zero, and once the crossover probability exceeds the upper bound of an experimentally obtained “good range”. This good range of p_c values depends on the population size: in *Table 1*, the speed of convergence starts rapidly decreasing once p_c exceeds about $0.50 - 0.60$, while, in *Table 3*, this phenomenon is observed once p_c exceeds $0.35 - 0.40$. We point out that the good or optimal range for p_c , in addition to the population size, is also dependent on the mutation rate, p_m . While much of the *GA* literature recommends p_c roughly within the range $[0.60, 0.90]$ (see, e.g., §3 in [3]), in all of our experiments (including many not presented herein), and for all mutation rates we have tested (roughly from 10% down to 0.1%), the fastest convergence has been typically obtained for values of p_c well below 0.60. Different population sizes and different values of p_m have been demonstrated to lead to different “flat regions” in p_c for which the convergence rates are at or around the optimum.

In the case of Problem 2, given the selection strategy and the mutation probability, the mean convergence rate is nearly constant with respect to changes in the crossover probability. This holds for all population sizes between 10 and 50 that we have experimented with. In case of larger populations (*Table 4*), the average speed of convergence is slightly improved by reducing the mutation probability p_m from 2% down to 0.1% (not shown above); however, the overall behavior remains just as “flat” with respect to, i.e., basically independent of, any changes in the crossover rate. Let us also notice that, in Problem 2, the variance across different experimental runs decreases considerably with an increase in the population size.

Another interesting observation is that, when the crossovers are required to produce offspring of the same string length as their parents, the performance in Problem 1 generally tends to deteriorate dramatically. On the other hand, letting some strings grow to about 10-12 genes in length apparently enables mutations and crossovers to efficiently evolve such longer “chromosomes” into the optimal solutions that contain all three desired patterns. Another possible “fix” for the slow convergence in Problem 1 is to consider the same goal patterns and symmetric crossovers only, but for (constant) string lengths greater than eight.

In another toy example, the goal was to make a population “learn” that optimal strings ought to contain both patterns *AAA* and *BBB* - but pattern *BAB* was considered irrelevant this time. Statistically, the performance for this, modified version of Problem 1 turned out to be rather similar to that for Problem 2 with the same corresponding

population sizes and values of p_m . In particular, “learning” *AAA + BBB* in the space of *fixed-length-eight* binary strings via *GA* rendered crossovers about just as irrelevant as learning *AAAA* did.

We conclude this brief and necessarily incomplete analysis with some remarks on the impact of the mutation rate, p_m . As expected, larger populations, in general, prefer lower p_m . However, this dependence is not linear. For both problems, changing p_m from 2% to 1% in case of larger population sizes had a bigger impact on the rate of convergence than reducing p_m from 1% down to 0.1%. Finding an optimal or close to optimal p_m is particularly important in case of the overall better behaved Problem 2, where the relative significance of mutations is higher, given the irrelevance of crossovers. Finally, in case of Problem 1, mutual dependence between p_c and p_m has been experimentally confirmed; due to space constraints, we leave the analysis of this dependence for another occasion.

4 Conclusions

Genetic algorithms are a useful computational paradigm for a variety of optimization and search problems. In particular, *GA* can be used effectively in many pattern recognition and pattern learning applications. However, even if a *GA* approach can be used for the given problem and a relatively fast convergence can be reached, we argue that this still does not imply that *GA* necessarily *should* be used in place of simpler search techniques. Whether a given problem truly warrants *GA* can be measured, in our view, by the relevance of the single uniquely *GA* operation, viz., the crossover. Thus, a problem that requires a non-negligible probability of crossovers and where the convergence rates are dependent on crossover rates deserves to be considered a worthy *GA* application. In contrast, a problem insensitive to crossover rates, and just as efficiently solvable by selection and mutations alone, as we see it, does not warrant the use of *GA*. We have illustrated these points with some simple pattern learning problems, seemingly quite similar to one another, yet where in some cases convergence critically depends on the crossover probability, while in others the choice of the crossover rate turns out to be practically irrelevant. Generalizing these simple examples and finding analytical characterizations of the two tentative classes of pattern learning problems appear to be worthy future research endeavors.

Acknowledgments: We are greatly indebted to Alfred Hubler (Center for Complex Systems Research), and we also sincerely thank Reza Ziae and Kirill Mechitov (Open Systems Laboratory), all from University of Illinois, for their assistance and feedback. The work presented herein was partly supported by the **DARPA IPTO TASK Program**, contract number **F30602-00-2-0586**.

Bibliography

- [1] Holland, J. H., *Genetic Algorithms and the Optimal Allocation of Trials*, SIAM J. Comp., 2, 1973
- [2] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989
- [3] Pal, S. H., P. P. Wang (ed.), *Genetic Algorithms for Pattern Recognition*, CRC Press, 1996
- [4] A. J. F. van Rooij, L. C. Jain, R. P. Johnson, *Neural Network Training Using Genetic Algorithms*, Machine Perception & Artificial Intelligence, vol. 26, World Scientific Publishing Co., 1996
- [5] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, 1992

AN ACTOR-BASED SIMULATION FOR STUDYING UAV COORDINATION

Myeong-Wuk Jang, Smitha Reddy, Predrag Tosic, Liping Chen, Gul Agha

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, IL 61801, USA

E-mail: { mjang, sreddy1, p-tosic, lchen2, agha } @cs.uiuc.edu

KEYWORDS

Actor, Simulation, Unmanned Aerial Vehicle (UAV), Coordination.

ABSTRACT

The effectiveness of Unmanned Aerial Vehicles (UAVs) is being increased to reduce the cost and risk of a mission [Doherty et al. 2000]. Since the advent of small sized but high performance UAVs, the use of a group of UAVs for performing a joint mission is of major interest. However, the development of a UAV is expensive, and a small error in automatic control results in a crash. Therefore, it is useful to develop and verify the coordination behavior of UAVs through software simulation prior to real testing. We describe how an actor-based simulation platform supports distributed simulators, and present three cooperation strategies: self-interested UAVs, sharing-based cooperation, and team-based coordination. Our experimental results show how communication among UAVs improves the overall performance of a collection of UAVs on a joint mission.

1. INTRODUCTION

The effectiveness of Unmanned Aerial Vehicles (UAVs) is being increased to reduce the cost and risk of a mission [Doherty et al. 2000]. Some military UAVs, such as the Predator and the Global Hawk, were already used during the wars in Afghanistan and Iraq. Decreasing size of the UAVs and increased demand for more intelligent and autonomous behavior of UAVs are paving the way for consideration of a group of UAVs performing a joint mission. While the cost of UAVs is lower than that of real planes, the development cost of a UAV is still very high, and a small error in automatic control may result in a crash. Therefore, when we consider a large number of UAVs working together, it is necessary to design and verify the behavior of UAVs through software simulation prior to real testing.

Many simulators have been developed as single process simulators. However, a single process simulator has several limitations. First, the performance of a simulation depends on the computational power of one computer. Second, a single process simulator has an extensibility issue when a special component requires its own specific process. For example, if we want to simulate the

coordination behavior of many virtual UAVs with a few real UAVs, each real UAV is working as an independent process. In this kind of simulation, a single process simulator cannot work well. Therefore, a concurrent object-based distributed simulator provides a better simulation environment.

It is commonplace to say that human beings are disposed to cooperate. Biology and ethology show that “kin-altruism” and “reciprocal-altruism” can ground cooperative behavior in animals, such as wolves surrounding prey, termites nest building, and birds flocking. Drawing a parallel, intelligent UAVs that cooperate with one another are of high interest for their ability to search, detect, identify, and handle targets together. The old age tenets of pre-planning and central control have to be reexamined, giving way to the idea of coordinated execution. In this paper, we describe and analyze three different strategies to coordinate tasks among UAVs in a dynamic environment to achieve their goals.

The outline of this paper is as follows. Section 2 sketches a simulation scenario and explains basic concepts about the actor model and the metrics in our simulation. Section 3 describes architecture for our simulation, and three cooperation strategies for a joint mission are presented in Section 4. Section 5 explains our implementation and experimental results. Then, in Section 6 and 7, we discuss related work and our future work. Finally, we conclude this paper with a summary of our simulation framework and our major contributions.

2. TERMINOLOGY

2.1 UAV Simulation Scenario

Prior to embarking on the architecture of our UAV simulator, we present a simple scenario in order to explain the meaning of basic terms. The application of our simulation is a UAV surveillance mission. For example, 50 UAVs might be launched into a certain area by *Ground Control System* (GCS) to detect targets in the area. For example, *targets* may be civilians to be rescued. In the simulation, UAVs have the autonomy to perform their mission without interaction with the GCS, except during the initial stage when message exchange is necessary to get each UAV started by sending them some default air routes. When UAVs are launched, the UAVs do not have any information about locations of targets. However, each

UAV is equipped with some sensors which can detect objects within the certain range. We assume that all UAVs start from the same location, called an *air base*. Controlling the sequence of takeoffs and landings of UAVs is managed by the control center, called *Air Base System* (ABS). The main task of a UAV is to detect locations of targets in a mission area and investigate them. Therefore, even though they navigate according to the given air routes, they can change their trajectories to handle targets once they detect those targets. In addition, when UAVs encounter *obstacles*, such as tall towers or airplanes, they should change their air routes to avoid a collision. Therefore, in our UAV simulation, there are five types of important components: Ground Control System (GCS), Air Base System (ABS), Unmanned Aerial Vehicles (UAVs), targets, and obstacles.

2.2 Actor

Our UAV simulator is based on the *Actor system*, a concurrent object-based distributed system, and hence, we use the actor model to describe each component in the simulation. An *actor* is a self-contained active object which has its own control thread and communicates with other actors through asynchronous message passing [Agha 1986; Agha et al. 1997]. In addition, an actor can create other actors, just as an object can create other objects. In our UAV simulator, each component, such as a UAV or a target, is implemented as an actor. Since these components in real situations operate concurrently and communicate with one another, their behavior can be captured very well by the actor model. Each software component in the simulation progresses its state independently of the progress of others in response to the environment information gathered either through its own sensor or by communicating with others.

2.3 Attractive Force Value and Utility Value

In our UAV simulation, each target has its own value. This value could be interpreted in several different ways. The value might correspond to the number of soldiers or the importance of a building. Also, we can consider this value as the time required to investigate a target by a UAV. For the simplicity of our simulation, we use a single numeric value instead of symbolic information or time information about a target.

In our simulation, we make the following assumptions. A UAV handles only one target at a time, although the UAV holds and manages information about several targets. In the advent of multiple targets to be handled, the UAV should select one of them. For this purpose, a UAV uses the attractiveness function to decide on a target. The *attractiveness function* maps the value of a target to the *attractive force value*, which represents a UAV's preference. This function depends on the value of the target and the distance between itself and the UAV, and is used to select the best target as follows:

$$\Theta_i(t) = \arg \max_j \left\{ \frac{\Pi_j(t)}{\|x_i(t) - \psi_j(t)\|} \right\}$$

where $\Pi_j(t)$ denotes the value of target j at time t , $x_i(t)$ is the location of UAV i at time t , and $\psi_j(t)$ is the location of target j at time t . If target j is stationary, $\psi_j(t)$ is always the same regardless of time. The value between braces is called the attractive force value of target j , and $\Theta_i(t)$ returns the index of the target that has the maximum attractive force value to UAV i at time t .

As a UAV approaches a target, the UAV starts consuming the value of the target once the UAV is within a certain distance of the target. We call the value consumed by the UAV the *utility value*. The *utility value function* and the *target value function* of the target i at time step $t+1$ are defined as follows:

$$U_i(t+1) = \Pi_i(0) - \Pi_i(t+1)$$

$$\Pi_i(t+1) = \max\{\Pi_i(t) - d \cdot n_i(t), 0\}$$

where $U_i(t)$ means the utility value of the target i at time t , d is a discount factor, and $n_i(t)$ is the number of UAVs which are near to the target i at time t . Therefore, in our simulation, when several UAVs are within the range of a target, the value of the target is consumed more quickly.

After a UAV reaches a target, it will fly around the target until the whole value of the target is consumed, either by the UAV alone or in conjunction with a group of UAVs. In our UAV simulation, one purpose of collective behavior of UAVs is to maximize the accumulated utility value within as short a time as possible. Here, the *accumulated utility value* means the whole value of targets consumed by all the UAVs.

3. SIMULATION ARCHITECTURE

Our distributed simulation is comprised of three layers: user interface, UAV simulator, and actor-based distributed platform (Figure 1). The *user interface layer* consists of two programs: Configuration Interface Program and Simulation Viewer. *Configuration Interface Program* provides an easy means of defining important attributes for the simulation. *Simulation Viewer* is a tool to check and verify the simulation results. All task oriented components, such as UAVs and targets, and simulation oriented components, such as Simulation Control Manager (SCM) and Active Broker (AB), are implemented as actors in the *UAV simulator layer*, which will be further explained in section 3.2.2. Each actor has its own thread to progress its state. The thread execution and communication of actors are

controlled by the Actor Foundry, an *actor-based distributed platform*.

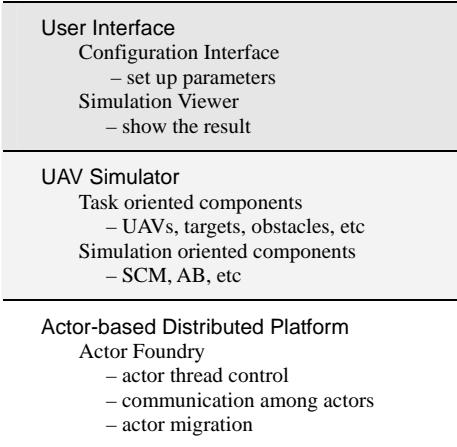


Figure 1: Three-layered Architecture for Distributed Simulation

The actor-based distributed platform is a middleware to support several distributed applications and is not tailor made for a specific simulation, such as a UAV simulation. The UAV simulator defines specific behaviors of UAVs, but does not include all the parameters to test and verify a behavior. These parameters are defined in user interface programs by a user and used in the UAV simulator. The functions of each layer are explained in detail below.

3.1 Actor-based Distributed Platform

The Actor Foundry is implemented in the Java programming language, and supports actor execution, communication between actors, and actor migration [Astley 1999; Clausen 1998].

In the Actor Foundry, an actor is created by another actor or by a user. When an actor is created, the actor name of the new actor is returned. This name would be used to refer to the receiver actor in message passing or deliver the reference of another actor to the receiver actor. The actor name is unique in the actor world. Therefore, even though an actor migrates from one host to another, the name is always transparent to other actors, and hence, other actors can continuously use the same name to refer to the given actor irrespective of that actor's current location, thereby providing a means for location transparency.

An actor in the Actor Foundry is running as a Java thread, and an actor communicates with other actors through asynchronous message passing. This is the main difference between the Actor Foundry and other object-based distributed platforms, such as CORBA and DCOM [Grimes 1997; OMG 2002]. In other object-based distributed platforms, one thread control is assumed: when an object is called by another object, the caller object is blocked until the called object returns the thread control. In the Actor Foundry, since every actor has its own control thread

to perform its operation and communicates with others through the asynchronous communication, the execution of an actor does not depend on those of others. Due to these features, we can easily use the power of distributed systems. Simulation components implemented as actors run on different computers independently, and they can communicate with others through the unique actor name, even though the distributed platform migrates some components from one host to another.

When distributed components interact with each other through asynchronous communication, analyzing the delivery sequence of communication messages is burdensome because asynchronous communication does not guarantee the message delivery order requirements, such as FIFO order, causal order, or total order [Hadzilacos and Toueg 1993]. Our distributed platform makes a log for message passing among actors, so that users can easily analyze the delivery sequence of messages.

3.2 UAV Simulator

All simulation components in our UAV Simulator can be classified into two categories: task oriented components and simulation oriented components (Figure 2). Task oriented components simulate objects in real situations. For example, a UAV component maps to a UAV object in a real situation while a target component maps to a target object. For the purpose of simulation, we need some virtual components, such as Simulation Control Manager and Active Broker. The following sub-sections explain these two categories of components in detail.

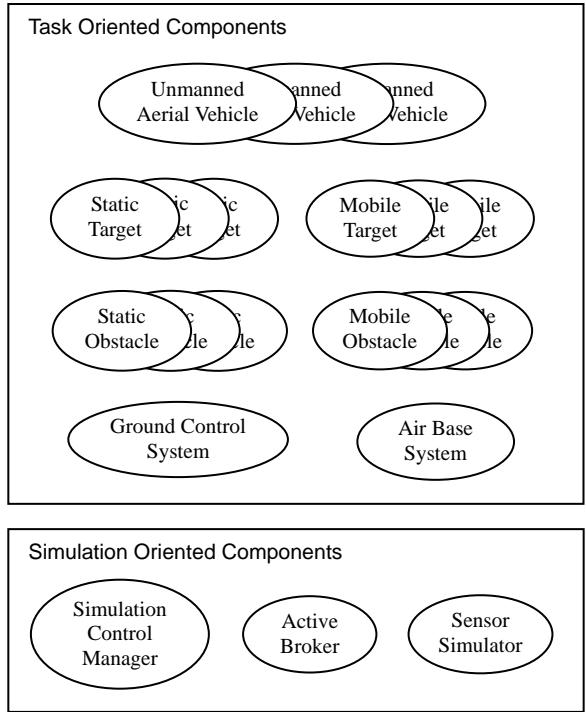


Figure 2: Simulation Components in UAV Simulator

3.2.1 Task Oriented Components

Task oriented components in our UAV simulator consist of five types: Ground Control System (GCS), Air Base System (ABS), Unmanned Aerial Vehicles (UAVs), obstacles, and targets. *GCS* is a central manager of UAVs and is aware of the mission area so as to indicate each UAV its air route in the area. However, GCS may not communicate continuously with UAVs to decide behavior of the UAVs at each time step because UAVs are supposed to perform their mission autonomously. *ABS* represents a control center of an air base and controls the sequence of take-offs and landings of UAVs. *UAVs* perform a given mission autonomously within certain restrictions, such as their kinematics and communication capability. *Obstacles* represent objects in which UAVs are not interested and with which a collision can happen. According to whether an obstacle can move or not, they are classified into two classes: *a mobile obstacle*, such as an airplane, and *a static obstacle*, such as a tall tower or a building. *Targets* represent objects of interest for the UAVs, such as, civilians to be rescued. According to its mobility characteristics, there are mobile targets and static targets.

3.2.2 Simulation Oriented Components

3.2.2.1 Simulation Control Manager

Each component manages its virtual time because each actor has its own control thread. However, this situation can cause inconsistency in virtual times of components. To maintain consistency between virtual times, *Simulation Control Manager* (SCM) manages local virtual times of the simulation components. When every component starts its execution, the initial value of each local virtual time is set to 0. After every component starts, SCM broadcasts a *virtual time clock message* to the other components. When a component receives the message, the component increases its local time and performs a small portion of its task that should be completed during the predefined time slice unit. For example, when a UAV receives the message, it updates its location and direction vector, and also checks whether or not new objects, such as other UAVs, targets, or obstacles, are detected. If a new neighboring UAV is detected, the UAV might exchange some information with the new neighboring UAV. After a component finishes its computation, it sends a reply message to SCM. When SCM receives reply messages from all the other components, SCM increases its virtual time, and rebroadcasts another virtual time clock message.

3.2.2.2 Active Broker

In order for a UAV to perform a group mission, the UAV needs to communicate with its neighboring UAVs through local broadcasting. *Active Broker* simulates a local broadcasting mechanism. In general, the brokering service supports attribute-based

communication. For example, if every UAV registers information about its current flying area with its actor name on a shared space, then when a UAV requests a broker for a message passing with a template that describes a certain area, the broker delivers the message to other UAVs which are in the area. However, this approach is not very accurate for finding the neighboring UAVs. Therefore, we have extended the function of the brokering service. In the active brokering service, every UAV registers information about its current location with its actor name on the shared space, and a UAV sends a special object instead of the template to request a message delivery to Active Broker. The object includes a specific method to be called by Active Broker. The method computes the distance between the location of the sender UAV and other UAVs and selects some which are within the local communication range. When the method returns actor names of neighboring UAVs, Active Broker delivers to them the message received from the sender UAV.

3.2.2.3 Sensor Simulator

Although each real UAV is supposed to be equipped with its own radar sensor, the radar sensors of all UAVs is simulated by a single simulation oriented component, *Sensor Simulator*. In the simulation, UAVs, targets, and obstacles register their current locations on a shared space every second in virtual time. Sensor Simulator periodically computes the distance between any two objects. If some components are within the sensor range of a UAV, Sensor Simulator reports information about these components to the UAV. Each UAV regards this information as its sensor input.

3.2.3 UAV Architecture

The most important simulation component is a UAV component. Therefore, we explain the architecture and the main behavior of a UAV in this subsection. A UAV is comprised of four modules: the physical process module, the trajectory planning module, the cooperative module, and the global control module (Figure 3).

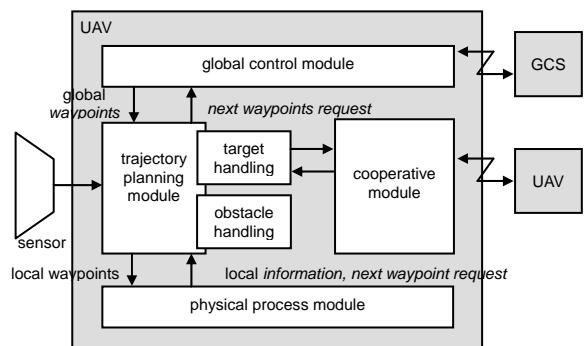


Figure 3: The Architecture of the Unmanned Aerial Vehicle Actor

When a UAV starts its mission, it does not have any information about its air route or the mission area. In our simulation, an air route is defined as a set of *waypoints* that need to be traversed by the UAV. Therefore, the first task of a UAV is to request the waypoints from GCS. The *global control module* of a UAV takes charge in communicating with GCS and managing the waypoints received. We call these waypoints *global waypoints*. When a UAV detects targets or/and obstacles, this information is delivered to the trajectory planning module from Sensor Simulator. The *trajectory planning module* handles them according to the predefined rules. For example, when a UAV detects several targets, it selects one target which has the best attractive force value, and then modifies its air route to reach the target. This function is performed by adding a waypoint to the list of UAV's current waypoints. The set of waypoints used inclusive of the additional waypoints are called *local waypoints*. The *cooperative module* is used when several UAVs want to handle a set of targets. To decide which UAV handles which target, the UAVs communicate with each other through the cooperative module. The kinematics of a UAV is implemented in the *physical process module*. Therefore, whenever this module receives a virtual time clock message, the physical process module computes the next location and the next direction of the UAV. When a UAV reaches the current waypoint, this module starts a turn toward the next waypoint according to the predefined kinematics.

3.3 User Interface

If we have to modify the UAV simulator whenever we execute it with different parameters, it is quite burdensome. Besides, modification at the code level requires comprehension making it hard for novice users to modify the code. In our architecture of UAV simulation, we separate the parameter modification part from the UAV simulator code as the user interface layer. Moreover, we separate the simulation checking part from simulator code. Therefore, the user interface layer consists of two programs: Configuration Interface Program and Simulation Viewer.

3.3.1 Configuration Interface Program

For the convenience of novice users, we have separated the configuration for UAV simulation parameters from the simulator code as a configuration file. This file can be modified by the Configuration Interface Program (Figure 4). Therefore, although a user does not look at and understand the source code for UAV simulation, they can change important parameters of simulation and run it without recompiling the source code. With this program a user can set up the number of UAVs, the size of mission area, the attributes of targets and obstacles, maximum simulation time, and the size of simulation time slice unit.

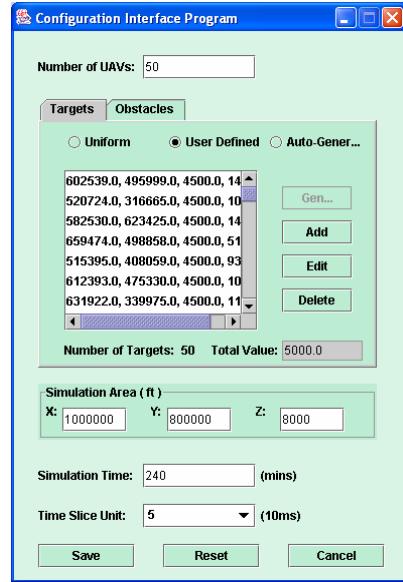


Figure 4: Configuration Interface Program

3.3.2 Simulation Viewer

Because of the characteristics of large scale simulations whose durations may sometimes be so long that we cannot monitor the simulation results continuously, we have separated the simulation checking from the simulation execution. Therefore, we look at and check the simulation results through Simulation Viewer (Figure 5). Another advantage of this approach is that the simulation results can be viewed back and forth with respect to the simulation virtual time.

While our UAV simulator is running according to the given parameters, the simulator generates simulation results on data files. The data files contain the locations and directions of UAVs, targets, and obstacles at every simulation virtual time step. The Simulation Viewer is used to check and verify the simulation results.

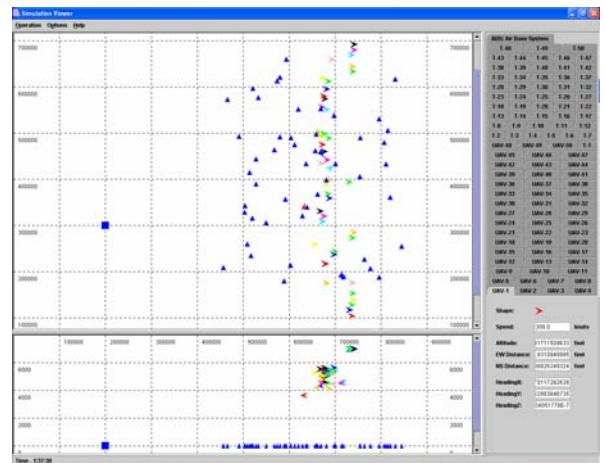


Figure 5: Simulation Viewer

4. COOPERATION AND COORDINATION AMONG UAVS

Cooperation among the UAVs is essential in directing the adjustment of policies in the globally most beneficial direction. In addition to cooperative dissemination of information, coordination of actions in larger teams is essential. With elements of uncertainty existing in the environment, coordination among UAVs has to be adaptive. The UAVs need to dynamically allocate responsibilities for different subtasks depending on the changing circumstances of the overall situation. For example, if additional targets are detected during a group mission, a team of UAVs needs to be able to handle them either by recruiting new member UAVs or changing the previous assignment of targets. In our UAV simulation, we use three strategies: the self-interested UAV strategy, the sharing-based cooperation strategy, and the team-based coordination strategy.

4.1 Self-interested UAVs

In the self-interested UAV strategy, a UAV senses a target and approaches it with the intention of consuming its entire value. When another UAV detects the same target, it also proceeds to consume the value of the target, irrespectively of what other UAVs do. Incessant polling of the target value till such time it is consumed completely serves as a means of interaction among the UAVs. It is not unusual to have more than one UAV concentrated on a target resulting in quicker consumption of its value, but also possibly in duplication of service.

4.2 Sharing-based Cooperation

In this strategy, once a UAV has discovered and located a target, it broadcasts this information so that other UAVs could direct their attention to the remaining targets. Reception of such information will result in the UAVs purging the targets that were advertised. This approach allows for a larger set of targets to be visited in a given time interval and is thus expected to be faster in accomplishing the mission goal. Exchange of information between UAVs referring to the same target will result in a UAV with a lower identification number to determine the UAV that would be responsible for this target based on parameters such as the distance from the target.

4.3 Team-based Coordination

In the team-based coordination strategy, certain UAV takes on the mantle of the leader of its team and dictates course of action to the other UAVs about the targets they need to visit. A team is dynamically formed and changed according to the set of targets detected; i.e. when a UAV detects more than one target, the UAV tries to handle the targets together with its neighboring UAVs. At this time, the main concern is

how to select an optimum UAV and decide the number of UAVs required to accomplish a task, when there are a sufficient number of neighboring UAVs. As the basic coordination protocol, we use the Contract Net protocol [Smith 1980; Smith and Davis 1981]. The UAV initiating the group mission works as the *group leader UAV*, and the other participant UAVs are called *member UAVs*. When a member UAV detects another target, the UAV delivers information about the new target to the leader UAV, and the leader UAV will add the target to the set of targets to be handled. The leader UAV considers the distance between a target detected and neighboring UAVs to assign the target. When a member UAV consumes the entire value of a target the UAV secedes from its group.

5. EXPERIMENTAL RESULT

We have developed the UAV simulator and two interface programs in Java programming language. Our UAV simulator is running on the Actor Foundry, but interface programs do not require the Actor Foundry. In order to simulate the flying and turning behavior of UAVs, we use the basic kinematics model of airplanes, but we abstract away the detailed dynamics and kinetics of aircraft.

For the UAV simulation, the size of the simulation area is set to $1,000,000 \times 800,000 \times 8,000$ cubic feet (length \times width \times altitude), size of the mission area to $400,000 \times 500,000 \times 8,000$ cubic feet, the radius of local broadcast communication of a UAV to 50,000 feet, and the radius of radar sensor to 25,000 feet. There are 50 targets in the mission area, and they are normally distributed. Half of the targets are static and the others are dynamic targets. When a UAV is within 1,000 feet from a target, the UAV consumes the value of the target. The initial value of each target is 100, and the discount factor d in the target value function is 5 per second.

To investigate how different cooperation strategies influence the performance of a joint mission, we use Average Service Cost (ASC) defined as follows:

$$ASC = \frac{\sum_i^n (NT_i - MNT)}{n}$$

where n is the number of UAVs, NT_i means navigation time of UAV i , MNT (Minimum Navigation Time) means average navigation time of all UAVs required for a mission when there are no targets.

Figure 6 shows Average Service Cost for three different cooperation strategies. When the number of UAVs is increased, ASC is decreased in every case. However, the sharing-based cooperation strategy and the team-based coordination strategy are better than the self-contained UAV strategy. From this result, we conclude that communication of UAVs is useful to handle targets, even though UAVs in the self-

contained UAV strategy consumes quickly the value of a target when they handle the target together. Another interesting result is the performance of the team-based coordination strategy is similar to that of the sharing-based cooperation strategy, even though the algorithm of the sharing-based cooperation strategy is much simpler. The overall ASC of the team-based coordination strategy is 3 or 5 seconds faster than that of the sharing-based cooperation strategy. When $n_i(t)$ in the target value function is not used, the performances of the sharing-based cooperation strategy and the team-based coordination strategy are not changed very much while that of the self-interested UAV strategy is decreased (Figure 7).

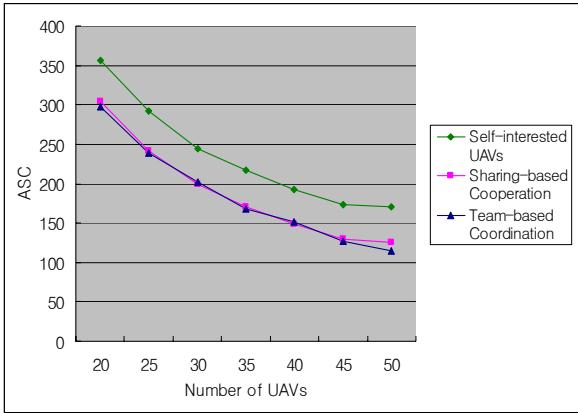


Figure 6: Average Service Cost (ASC) for three different coordination strategies.

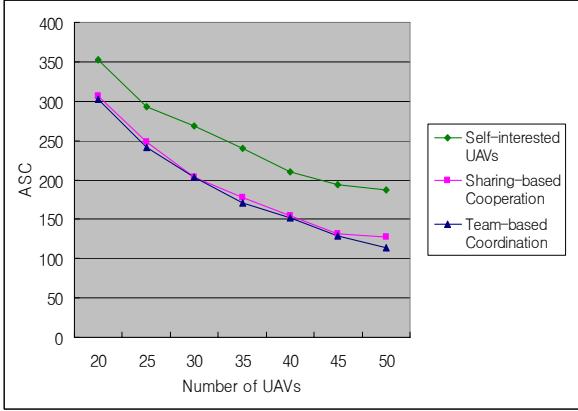


Figure 7: Average Service Cost when $n_i(t)$ is not used.

6. RELATED WORK

Johnson and Mishra present a flight simulation tool for GTMax (Georgia Tech R-Max VTOL UAV) [Johnson and Mishra 2002]. Barney Pell and his colleagues describe the NMRA (New Millennium Remote Agent), architecture for a UAV. The NMRA integrates real-time monitoring and control with planning and scheduling, handles fault recovery and reconfiguration of component models, and simulates the autonomy of a UAV [Pell et al. 1997]. However, the type of the

GTMax UAV is a helicopter, and both papers do not handle cooperation among UAVs.

Altenburg and his colleagues present an agent based simulator to simulate UAV cooperative control [Altenburg et al. 2002]. In their approach, agents are reactive agents while UAV components in our simulation are deliberative agents. Therefore, their agents directly respond to signals from environment, while our agents change their intention about targets and automatically and proactively select a different action. Also, their agents communicate with others indirectly through the environment while our agents communicate with each others directly. Kolek and his colleagues present a simulation framework to evaluate the performance of real time tactical radio networks with a UAV [Kolek et al. 1998]. In this paper, the authors explain how much distributed simulation could be applied to solve military problems, but they do not handle the autonomy of UAVs and coordination among UAVs.

7. FUTURE WORK

The Actor system supports distributed computational environment and actor mobility. In the current platform, it is the programmer's role to determine actor placement. However, this is hard to do when we do not know the CPU speed and the communication speed among different machines. Specifically, when the communication pattern among actors is changed, the initial placement of actors might prove to be a deterrent to cross boundary communication. For this, we are developing dynamic actor reconfiguration algorithm. In the new actor platform, the communication pattern among actors will be monitored, and actors will be dynamically reallocated by the platform.

Another problem of the current actor system is the existence of Simulation Control Manager (SCM) to control the virtual times of UAVs globally. This component may be a bottleneck of the distributed simulation, and if this component were to fail, the simulation would collapse completely. To counter this, the Jefferson's virtual time [Jefferson 1985] based actor platform can be used. In this actor platform, each actor maintains its own virtual time, and when an actor communicates with another actor and the time difference is more than the given threshold, the platform performs the rollback.

As another extension, we are looking to merge a few real UAVs into UAV simulation. That is, we are going to build a UAV simulator with the possibility of real time input from real UAVs and virtual UAVs. In this simulation, a real UAV can communicate with other real UAVs and virtual UAVs to perform a virtual task. This approach can overcome the problem of computer simulation, such as the inaccuracy of UAV kinematics and the communication delay defined by programmers.

In our simulation, we use Contract Net Protocol. It means if a UAV accepts the order from a leader UAV,

the UAV must handle the target. However, the belief about environment changes when UAVs detects more targets or additional UAVs become available after having consumed value of their respective targets. Therefore, when any change in the environment is detected or any UAV becomes available, this information is delivered to the leader UAV, and the leader UAV may reconsider and change the target assignment. Also, a member UAV may secede from its team to handle a new target with a more attractive force value. This idea is motivated from the leveled commitment in Contract Net Protocol [Sandholm and Lesser 1995].

8. CONCLUSIONS

In this paper, we have described the design and development of a distributed UAV simulator using an actor-based platform, a utility function, and Contract Net Protocol. The three layered architecture for our UAV simulation is presented: the actor-based distributed platform, the UAV simulator, and the user interface layer. We have described three strategies to perform a joint mission: the self-interested UAVs strategy, the sharing-based coordination strategy, and team-based cooperation strategy. This has been supplemented by our experimental results and outline of the future work.

Our UAV simulator is working on an actor-based distributed platform, and hence, it naturally adapts to the behavior of a distributed and concurrent situation. We can easily improvise the execution environment without changing the UAV simulator by separating the distributed platform from the simulator. For example, we can migrate some actors from a computer to another during the execution time. Other possible means for improvising the working environment have been presented in the future work section. When we consider multiple UAVs working together, their cooperation mechanisms are of utmost importance. In this paper, we have presented three different approaches, and compared and contrasted them. The experimental results suggest that cooperation and coordination strategies are better than the self-interested UAV strategy. Last but not least, we have introduced the active brokering service to support application oriented searching.

ACKNOWLEDGEMENT

This research is sponsored by the Defense Advanced Research Projects Agency under contract number F30602-00-2-0586. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

REFERENCES

Agha, G.A. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass.

Agha G.A.; I.A. Mason; S.F. Smith; and C.L. Talcott. 1997. "A Foundation for Actor Computation." *Journal of Functional Programming*, Vol. 7, No. 1, 1-69.

Altenburg K.; J. Schlecht; and K. Nygard. 2002. "An Agent-based Simulation for Modeling Intelligent Munitions." In *Proceedings of the Second WSEAS International Conference on Simulation, Modeling and Optimization*, Skiathos, Greece (Sep). Available at <http://www.cs.ndsu.nodak.edu/~nygard/research/munitions.pdf>

Astley M. 1999. *Actor Foundry*. Department of Computer Science, University of Illinois at Urbana-Champaign, IL (Feb. 9). Available at <http://yangtze.cs.uiuc.edu/foundry>

Clausen T.H. 1998. *Actor Foundry – a QuickStart*. Department of Computer Science, Institute of Electronic Systems, Denmark (Nov. 9). Available at <http://yangtze.cs.uiuc.edu/foundry>

Doherty P.; G. Granlund; K. Kuchcinski; E. Sandewall; K. Nordberg; E. Skarman; and J. Wiklund. 2000. "The WITAS Unmanned Aerial Vehicle Project." In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000)*, Berlin, Germany (Aug), 747-755.

Grimes R. 1997. *Professional DCOM Programming*. Olton, Birmingham, Canada, Wrox Press.

Hadzilacos V. and S. Toueg. 1993. "Fault-Tolerant Broadcasting and Related Problems." In *Distributed Systems*, S. Mullender (Ed.). ACM Press, New York, 97-145.

Jefferson D. 1995. "Virtual Time." *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3 (Jul), 404-425.

Johnson E.N and S. Mishra. 2002. "Flight Simulation for the Development of an Experimental UAV." In *Proceeding of the AIAA Modeling and Simulation Technologies Conference and Exhibit*, Monterey California, CA (Aug), 5-8.

Kolek S.R.; S.J. Rak; and P.J. Christensen. 1998. "Battlefield Communication Network Modeling." *The DIS Workshop on Simulation Standards*. Available at <http://dss.ll.mit.edu/dss.web/98F-SIW-143.html>

OMG 2002. *The Common Object Request Broker Architecture: Core Specification*. Version 3.0.2 (Dec).

Pell B.; D.E. Bernard; S.A. Chien; E. Gat; N. Muscettola; P.P. Nayak; M.D. Wagner; and B.C. Williams. 1997. "An Autonomous Spacecraft Agent Prototype." In *Proceedings of the First International Conference on Autonomous Agents*, Marina del Rey, CA, 253-261.

Sandholm T. and V. Lesser. 1995. "Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Framework." In *Proceedings of the 1st International Conference on Multiagent Systems*, San Francisco, CA, 328-335.

Smith R.G. 1980. "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver." *IEEE Transactions on Computers*, Vol. 29, No. 12, 1104-1113.

Smith R.G. and R. Davis. 1980. "Frameworks for Cooperation in Distributed Problem Solving." *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 11, No. 1, 61-70.

AUTHOR BIOGRAPHIES

MYEONG-WUK JANG is a doctoral candidate and research assistant in the Open Systems Laboratory at the University of Illinois at Urbana-Champaign. His research interests include multi-agent system and task allocation in open distributed computing. He received a BS in Computer Science from Korea University in 1990 and an MS in Computer Science from KAIST (Korea Advanced Institute of Science and Technology) in 1992. He worked for ETRI (Electronics and Telecommunications Research Institute), Korea, until 1998. His web page can be found at <http://www.uiuc.edu/~mjjang/>.

SMITHA REDDY is a Master/PhD student and research assistant in the Open Systems Laboratory at the University of Illinois at Urbana-Champaign. Her research interests include distributed systems, high speed networks, and dynamic resource sharing. She received a BE in Computer Science from University of Pune in 1999.

PREDRAG TOSIC is a doctoral candidate and research assistant in the Open Systems Laboratory at the University of Illinois at Urbana-Champaign. He received a BS in Mathematics and Physics and an MS in Applied Mathematics, both at University of Maryland Baltimore County, UMBC, in 1994 and 1995, respectively, and also holds an MS in pure Mathematics from University of Illinois at Urbana-Champaign in 1997.

LIPING CHEN is a doctoral candidate and research assistant in the Open Systems Laboratory at the University of Illinois at Urbana-Champaign.

GUL A. AGHA is Director of the Open Systems Laboratory at the University of Illinois at Urbana-Champaign and Professor in the Department of Computer Science. His research interests include models, languages, and tools for parallel computing and open distributed systems. He received a BS in an interdisciplinary program from the California Institute of Technology, an MA in Psychology from the University of Michigan, Ann Arbor, and an MS and PhD in Computer and Communication Science, from the University of Michigan, Ann Arbor.

Update History:

We have corrected an error in the utility value function and reduced the size of arrows in Figure 3.

- November 19, 2003

Appendix AD:

A Rewriting Based Model for Probabilistic Distributed Object Systems

Nirman Kumar, Koushik Sen, José Meseguer, Gul Agha

Department of Computer Science,

University of Illinois at Urbana-Champaign.

{nkumar5,ksen,meseguer,agha}@cs.uiuc.edu

Abstract. Concurrent and distributed systems have traditionally been modelled using nondeterministic transitions over configurations. The nondeterminism provides an abstraction over scheduling, network delays, failures and randomization. However a probabilistic model can capture these sources of nondeterminism more precisely and enable statistical analysis, simulations and reasoning. We have developed a general semantic framework for probabilistic systems using probabilistic rewriting. Our framework also allows nondeterminism in the system. In this paper, we briefly describe the framework and its application to concurrent object based systems such as actors. We also identify a sufficiently expressive fragment of the general framework and describe its implementation. The concepts are illustrated by a simple client-server example.

Keywords: Rewrite theory, probability, actors, Maude, nondeterminism.

1 Introduction

A number of factors, such as processor scheduling and network delays, failures, and explicit randomization, generally result in nondeterministic execution in concurrent and distributed systems. A well known consequence of such nondeterminism is an exponential number of possible interactions which in turn makes it difficult to reason rigorously about concurrent systems. For example, it is infeasible to use techniques such as model checking to verify any large-scale distributed systems. In fact, some distributed systems may not even have a finite state model: in particular, *networked embedded systems* involving *continuously* changing parameters such as time, temperature or available battery power are infinite state.

We believe that a large class of concurrent systems may become amenable to a rigorous analysis if we are able to quantify some of the probabilities of transitions. For example, network delays can be represented by variables from a probabilistic distribution that depends on some function of the system state. Similarly, available battery power, failure rates, etc., may also have a probabilistic behavior. A probabilistic model can capture the statistical regularities in such systems and enable us to make probabilistic guarantees about its behavior.

We have developed a model based on rewriting logic [11] where the rewrite rules are enriched with probability information. Note that rewriting logic provides a natural model for object-based systems [12]. The local computation of each object is modelled by rewrite rules for that object and one can reason about the global properties that result from the interaction between objects: such interactions may be asynchronous as in actors, or synchronous as in the π -calculus. In [9] we show how several well known models of probabilistic and nondeterministic systems can be expressed as special cases of probabilistic rewrite theories. We also propose a temporal logic to express properties of interest in probabilistic systems. In this paper we show how probabilistic object systems can be modelled in our framework. Our *probabilistic rewriting* model is illustrated using a client-server example. The example also shows how nondeterminism, for which we do not have the probability distributions, is represented naturally in our model. Nondeterminism is eventually removed by the system *adversary* and converted into probabilities in order to define a probability space over computation paths.

The *Actor* model of computation [1] is widely used to model and reason about object-based distributed systems. Actors have previously been modelled as rewrite theories [12]. Probabilistic rewrite theories can be used to model and reason about actor systems where actors may fail and messages may be dropped or delayed and the associated probability distributions are known (see Section 3).

The rest of this paper is organized as follows. Section 2 provides some background material on membership equational logic [13] and rewriting [11] as well as probability theory. Section 3 starts by giving an intuitive understanding of how a step of computation occurs in a probabilistic rewrite theory. We then introduce an example to motivate the modelling power of our framework and formalize the various concepts. In Section 4 we define an important subclass of probabilistic rewrite theories, and in Section 5, we describe its Maude implementation. The final section discusses some directions for future research.

2 Background and Notation

A *membership equational theory* [13] is a pair (Σ, E) , with Σ a *signature* consisting of a set K of *kinds*, for each $k \in K$ a set S_k of *sorts*, a set of *operator* declarations of the form $f : k_1 \dots k_n \rightarrow k$, with $k, k_1, \dots, k_n \in K$ and with E a set of *conditional Σ -equations* and Σ -*memberships* of the form

$$\begin{aligned} (\forall \vec{x}) t = t' &\Leftarrow u_1 = v_1 \wedge \dots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m \\ (\forall \vec{x}) t : s &\Leftarrow u_1 = v_1 \wedge \dots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m \end{aligned}$$

The \vec{x} denote *variables* in the terms t, t', u_i, v_i and w_j above. A membership $w : s$ with w a Σ -term of kind k and $s \in S_k$ asserts that w has sort s . Terms that do not have a sort are considered *error* terms. This allows membership equational theories to specify partial functions within a total framework. A Σ -*algebra* B consists of a K -indexed family of sets $X = \{B_k\}_{k \in K}$, together with

1. for each $f : k_1 \dots k_n \rightarrow k$ in Σ a function $f_B : B_{k_1} \times \dots \times B_{k_n} \rightarrow B_k$

- for each $k \in K$ and each $s \in S_k$ a subset $B_s \subseteq B_k$.

We denote the algebra of terms of a membership equational theory by T_Σ . The *models* of a membership equational theory (Σ, E) are those Σ -algebras that satisfy the equations E . The inference rules of membership equational logic are *sound* and *complete* [13]. Any membership equational theory (Σ, E) has an *initial algebra* of terms denoted $T_{\Sigma/E}$ which, using the inference rules of membership equational logic and assuming Σ *unambiguous* [13], is defined as a quotient of the term algebra T_Σ by

$$\begin{aligned} \bullet \quad t \equiv_E t' &\Leftrightarrow E \vdash (\forall \emptyset) t = t' \\ \bullet \quad [t]_{\equiv_E} \in T_{\Sigma/E,s} &\Leftrightarrow E \vdash (\forall \emptyset) t : s \end{aligned}$$

In [2] the usual results about *equational simplification*, *confluence*, *termination*, and *sort-decreasingness* are extended in a natural way to membership equational theories. Under those assumptions a membership equational theory can be executed by equational simplification using the equations from left to right, perhaps modulo some *structural* (e.g. associativity, commutativity and identity) axioms A . We denote the algebra of terms simplified by equations and structural axioms as $T_{\Sigma,E \cup A}$ and the isomorphic algebra of equivalence classes modulo axioms A , of equationally simplified terms by $Can_{\Sigma,E/A}$. The notation $[t]_A$ represents the equivalence class of a term t fully simplified by the equations.

In a standard *rewrite theory* [11], transitions in a system are described by labelled rewrite rules of the form

$$l : t(\vec{x}) \longrightarrow t'(\vec{x}) \text{ if } C(\vec{x})$$

Intuitively, a rule of this form specifies a *pattern* $t(\vec{x})$ such that if some fragment of the system's state matches that pattern and satisfies the condition C , then a local transition of that state fragment, changing into the pattern $t'(\vec{x})$ can take place. In a *probabilistic rewrite rule* we add probability information to such rules. Specifically, our proposed probabilistic rules are of the form,

$$l : t(\vec{x}) \longrightarrow t'(\vec{x}, \vec{y}) \text{ if } C(\vec{x}) \text{ with probability } \pi(\vec{x}).$$

In the above, the set of variables in the left-hand side term $t(\vec{x})$ is \vec{x} , while some new variables \vec{y} may be present in the term $t'(\vec{x}, \vec{y})$ on the right-hand side. Of course it is not necessary that *all* of the variables in \vec{x} occur in $t'(\vec{x}, \vec{y})$. The rule will match a state fragment if there is a substitution θ for the variables \vec{x} that makes $\theta(t)$ equal to that state fragment and the condition $\theta(C)$ is true. Because the right-hand side $t'(\vec{x}, \vec{y})$ may have new variables \vec{y} , the next state is not *uniquely* determined: it depends on the choice of an additional substitution ρ for the variables \vec{y} . The choice of ρ is made according to the probability function $\pi(\theta)$, where π is not a fixed probability function, but a *family* of functions: one for each match θ of the variables \vec{x} .

The Maude system [4,5] provides an execution environment for membership equational and rewrite theories. The Full Maude [6] library built on top of the Core Maude environment allows users to specify object oriented modules in a

convenient syntax. Several examples in [12,5] show specifications of object based systems in Maude. The code for our example in Section 3 is written in the syntax of Maude 2.0 [5].

To succinctly define probabilistic rewrite theories, we use a few basic notions from axiomatic probability theory. A σ -algebra on a set X is a collection \mathcal{F} of subsets of X , containing X itself and closed under complementation and finite or countably infinite unions. For example the power set $\mathcal{P}(X)$ of a set X is a σ -algebra on X . The elements of a σ -algebra are called *events*. We denote by $\mathcal{B}_{\mathbb{R}}$ the smallest σ -algebra on \mathbb{R} containing the sets $(-\infty, x]$ for all $x \in \mathbb{R}$. We also remind the reader that a *probability space* is a triple (X, \mathcal{F}, π) with \mathcal{F} a σ -algebra on X and π a *probability measure function*, defined on the σ -algebra \mathcal{F} which evaluates to 1 on X and distributes by addition over finite or countably infinite union of disjoint events. For a given σ -algebra \mathcal{F} on X , we denote by $PFun(X, \mathcal{F})$ the set

$$\{\pi \mid (X, \mathcal{F}, \pi) \text{ is a probability space}\}$$

Definition 1 (\mathcal{F} -cover). For a σ -algebra \mathcal{F} on X , an \mathcal{F} -cover is a function $\alpha : X \rightarrow \mathcal{F}$, such that $\forall x \in X \ x \in \alpha(x)$.

Let π be a probability measure function defined on a σ -algebra \mathcal{F} on X , and suppose α is an \mathcal{F} -cover. Then notice that $\pi \circ \alpha$ naturally defines a function from X to $[0, 1]$. Thus, for example, for $X = \mathbb{R}$ and $\mathcal{F} = \mathcal{B}_{\mathbb{R}}$, we can define α to be the function that maps the real number x to the set $(-\infty, x]$. With X a finite set and $\mathcal{F} = \mathcal{P}(X)$, the power set of X , it is natural to define α to be the function that maps $x \in X$ to the singleton $\{x\}$.

3 Probabilistic Rewrite Theories

A probabilistic rewrite theory has an interleaving execution semantics. A step of computation changes a term $[u]_A$ to $[v]_A$ by the application of a single rewrite rule on some subterm of the given *canonical* term $[u]_A$. Recall the form of a probabilistic rewrite rule as described in the previous section. Firstly, all context, rule, substitution (for the variables \vec{x}) triples arising from possible applications of rewrite rules (see definition 6) to $[u]_A$ are computed. One of them $([\mathbb{C}]_A, r, [\theta]_A)$ (for the justification of the A subscript see definitions 2, 3 and 5) is chosen *nondeterministically*. This step essentially represents the nondeterminism in the system. After that has been done, a particular substitution $[\rho]_A$ is chosen *probabilistically* for the new variables \vec{y} and $[\rho]_A$ along with $[\theta]_A$, is applied to the term $t'(\vec{x}, \vec{y})$ and placed inside the context \mathbb{C} to obtain the term $[v]_A$. The choice of the new substitution $[\rho]_A$ is from the set of possible substitutions for \vec{y} . The probabilities are defined as a *function* of $[\theta]_A$. This gives the framework great expressive power. Our framework can model both nondeterminism and probability in the system. Next we describe our example, model it as an object based rewrite theory and indicate how the rewrite rules model the probabilities and nondeterminism.

```

pmod QOS-MODEL is
  ...
  vars L N m1 m2: Nat.
  vars Cl Sr Nw: Oid.
  var i: Bit.
  vars C Q: Configuration.
  var M: Msg.
  op _ ← _: Oid Nat → Msg.
  class Client |sent:Nat, svc1:Nat, svc2:Nat.
  class Network |soup:Configuration.
  class Server |queue:Configuration.
  ops H Nt S1 S2: → Oid.
  ops acq1 acq2: → Msg.
  prl [req]:⟨Cl:Client|sent:N, svc1:m1, svc2:m2⟩⟨Nw: Network|soup:C⟩⇒
    ⟨Cl: Client|sent:(N + 1), svc1:m1, svc2:m2⟩⟨Nw: Network|soup:C (Sr ← L)⟩.
  cprl [acq]:⟨Cl:Client|svc1:m1, svc2:m2⟩⟨Nw:Network|soup:M C⟩⇒
    ⟨Cl:client|svc1:m1 + δ(i, M, 1), svc2:m2 + δ(i, M, 2)⟩⟨Nw:Network|soup:C⟩
    if acq(M).
  prl [deliver]:⟨Nw:Network|soup:(Sr ← L) C⟩⟨Sr:Server|queue:Q⟩⇒
    ⟨Nw:Network|soup:C⟩⟨Sr:Server|queue:Q M⟩.
  prl [process]:⟨Sr:Server|queue:(Sr ← L) Q⟩⟨Nw:Network|soup:C⟩ ⇒
    ⟨Sr:Server|queue:Q⟩⟨Nw:Network|soup:C M⟩.
endpm

```

Fig. 1. A client-server example

A Client-Server Example : Our example is a situation where a client is sending computational jobs to servers across a network. There are two servers S_1 and S_2 . S_1 is computationally more powerful than S_2 , but the network connectivity to S_2 is better (more reliable) than that to S_1 and packets to S_1 may be dropped without being delivered, more frequently than packets to S_2 . The servers may also drop requests if the load increases beyond a certain threshold. The computationally more powerful server S_1 drops packets with a lower probability than S_2 . We would like to reason about a good randomized policy for the client. The question here is: which server is it better to send packets to, so that a larger fraction of packets are processed rather than dropped? Four objects model the system. One of them, the client, sends packets to the two server objects deciding probabilistically before each send which server to send the packet to. The other object models a network, which can either transmit the packets correctly, drop them or deliver them out of order. The remaining two objects are server objects which either drop a request or process it and send an acknowledgement message. The relevant fragment of code specifying the example is given in Figure 1. The client object named H maintains the total number of requests sent in a variable $sent$ and those which were successfully processed by servers S_1, S_2 in variables svc_1 and svc_2 respectively. Notice that for example in $svc_1 : m_1$ the m_1 is the *value* of the variable named svc_1 . An example term representing a possible system state is

$$\langle H : \text{Client} \mid \text{sent} : 3, \text{svc}_1 : 1, \text{svc}_2 : 0 \rangle \langle Nt : \text{Network} \mid \text{soup} : (S_1 \leftarrow 10) \rangle$$

$$\langle S_1 : \text{Server} \mid \text{queue} : \text{nil} \rangle \langle S_2 : \text{Server} \mid \text{queue} : (S_2 \leftarrow 5) \rangle$$

The term above of sort *Configuration* (collection of objects and messages) represents a multiset of objects combined with an empty syntax (juxtaposition) multiset union operator that is declared associative and commutative. The client has sent 3 requests in total, out of which one has already been serviced by S_1 , one is yet to be delivered and one request is yet pending at the server S_2 . The numbers 10 and 5 represent the measure of the loads in the respective requests.

We discuss the rules labelled *req* and *acq*. Henceforward we refer to a rule by its label. Though not shown in Figure 1, a probabilistic rewrite theory associates some functions with the rules, defining the probabilities. The rule *req* models the client sending a request to one of the servers by putting a message into the network object's variable *soup*. The rule involves two new variables *Sr* and *L* on the right-hand side. *Sr* is the name of the server to which the request is sent and *L* is the message *load*. A probability function $\pi_{\text{req}}(Cl, N, m_1, m_2, Nw, C)$ associated with the rule *req* (see definition 4) will decide the distribution of the new variables *Sr* and *L*, and thus the randomized policy of the client. For example, it can assign higher probability values to substitutions with $Sr = S_1$, if it finds that $m_1 > m_2$; this would model a heuristic policy which sends more work to the server which is performing better. In this way the probabilities can depend on the values of m_1, m_2 (and thus the state of the system). In the rule labelled *acq* there is only one new variable *i* on the right-hand side. That variable can only assume two values 0, 1 with nonzero probability. 0 means a message drop, so that $\delta(0, M, 1) = \delta(0, M, 2) = 0$, while if $i = 1$ then the appropriate *svc* variable is incremented. The distribution of *i* as decided by the function $\pi_{\text{acq}}(\dots, M)$ could depend on *M*, effectively modelling the network connectivity. The network drops messages more frequently for $M = \text{acq}_1$ (an *acq* message from server S_1) than it does for $M = \text{acq}_2$. Having the distribution of new variables *depend* on the substitution gives us the ability to model general distributions. The associativity and commutativity attribute of the juxtaposition operator for the sort *Configuration* essentially allows nondeterminism in the order of message delivery by the network (since it chooses a message to process, from the associative commutative *soup* of messages) and the order of messages processed by the servers.

The more frequently the rewrite rules for the network object are applied (which allow it to process the messages in its *soup*), the more frequently the *acq* messages will be delivered. Likewise, the more frequently the rewrite rules for a particular server are applied, the more quickly will it process its messages. Thus, during a computation the values m_1, m_2 , which determine the client's randomized policy, will actually depend not only on the probability that a server correctly processes the packets and the network correctly delivers requests and acknowledgments, but also on how frequently the appropriate rewrite rules are applied. However, the exact frequency of application depends on the *nondeterministic* choices made. We can now see how the nondeterminism effectively influences the probabilities in the system. As explained later, the nondetermin-

ism is removed (converted into probabilities) by what is called an *adversary* of the system. In essence the adversary is like a *scheduler* which determines the rate of progress of each component. The choice of adversary is important for the behavior of the system. For example, we may assume a *fair* adversary that chooses between its nondeterministic choices equally frequently. At an intuitive level this would mean that the different parts of the system compute at the same rate. Thus, it must be understood that the model defined by a probabilistic rewrite theory is parameterized on the adversary. The system modeler must define the adversary based on an understanding of how frequently different objects in the system advance. Model checking of probabilistic systems quantifies over adversaries, whereas a simulation has to fix an adversary.

We now define our framework formally.

Definition 2 (E/A -canonical ground substitution). An E/A -canonical ground substitution is a substitution $\theta : \vec{x} \rightarrow T_{\Sigma, E \cup A}$.

Intuitively an E/A -canonical ground substitution represents a substitution of ground terms from the term algebra T_Σ for variables of the corresponding sorts, so that all of the terms have already been reduced as much as possible by the equations E and the structural axioms A . For example the substitution 10×2 to a variable of sort *Nat* is not a canonical ground substitution, but a substitution of 20 for the same variable is a canonical ground substitution.

Definition 3 (A -equivalent substitution). Two E/A -canonical ground substitution $\theta, \rho : \vec{x} \rightarrow T_{\Sigma, E \cup A}$ are A -equivalent if and only if $\forall x \in \vec{x} \ [\theta(x)]_A = [\rho(x)]_A$.

We use $\text{CanGSubst}_{E/A}(\vec{x})$ to denote the set of all E/A -canonical ground substitutions for the set of variables \vec{x} . It is easy to see that the relation of A -equivalence as defined above is an equivalence relation on the set $\text{CanGSubst}_{E/A}(\vec{x})$. When the set of variables \vec{x} is understood, we use $[\theta]_A$ to denote the equivalence class containing $\theta \in \text{CanGSubst}_{E/A}(\vec{x})$.

Definition 4 (Probabilistic rewrite theory). A probabilistic rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E \cup A, R, \pi)$, with $(\Sigma, E \cup A, R)$ a rewrite theory with the rules $r \in R$ of the form

$$l : t(\vec{x}) \rightarrow t'(\vec{x}, \vec{y}) \text{ if } C(\vec{x})$$

where

- \vec{x} is the set of variables in t .
- \vec{y} is the set of variables in t' that are not in t . Thus t' might have variables coming from the set $\vec{x} \cup \vec{y}$ but it is not necessary that all variables in \vec{x} occur in t' .
- C is a condition of the form $(\bigwedge_j u_j = v_j) \wedge (\bigwedge_k w_k : s_k)$, that is, C is a conjunction of equations and memberships;

and π is a function assigning to each rewrite rule $r \in R$ a function

$$\pi_r : \llbracket C \rrbracket \rightarrow P\text{Fun}(CanGSubst_{E/A}(\vec{y}), \mathcal{F}_r)$$

where $\llbracket C \rrbracket = \{[\mu]_A \in CanGSubst_{E/A}(\vec{x}) \mid E \cup A \vdash \mu(C)\}$ is the set of E/A -canonical substitutions for \vec{x} satisfying the condition C , and \mathcal{F}_r is a σ -algebra on $CanGSubst_{E/A}(\vec{y})$. We denote a rule r together with its associated function π_r , by the notation

$$l : t(\vec{x}) \rightarrow t'(\vec{x}, \vec{y}) \text{ if } C(\vec{x}) \text{ with probability } \pi_r(\vec{x})$$

We denote the class of probabilistic rewrite theories by **PRwTh**. Notice the following points in the definition

1. Rewrite rules may have new variables \vec{y} on the right-hand side.
2. The condition $C(\vec{x})$ on the right-hand side depends only on the variables \vec{x} occurring in the term $t(\vec{x})$ on the left-hand side.
3. The condition $C(\vec{x})$ is simply a conjunction of equations and memberships (but no rewrites).
4. $\pi_r(\vec{x})$ specifies, for each substitution θ for the variables \vec{x} , the probability of choosing a substitution ρ for the \vec{y} . In the next section we explain how this is done.

3.1 Semantics of Probabilistic Rewrite Theories

Let $\mathcal{R} = (\Sigma, E \cup A, R, \pi)$ be a probabilistic rewrite theory such that:

1. E is confluent, terminating and sort-decreasing modulo A [2].
2. the rules R are coherent with E modulo A [4].

We also assume a choice for each rule r of an \mathcal{F}_r -cover $\alpha_r : CanGSubst_{E/A}(\vec{y}) \rightarrow \mathcal{F}_r$. This \mathcal{F}_r -cover will be used to assign probabilities to rewrite steps. Its choice will depend on the particular problem under consideration.

Definition 5 (Context). A context \mathbb{C} is a Σ -term with a single occurrence of a single variable, \odot , called the hole. Two contexts \mathbb{C} and \mathbb{C}' are A -equivalent if and only if $A \vdash (\forall \odot) \mathbb{C} = \mathbb{C}'$.

Notice that the relation of A -equivalence for contexts as defined above, is an equivalence relation on the set of contexts. We use $[\mathbb{C}]_A$ for the equivalence class containing context \mathbb{C} . For example the term

$$\begin{aligned} & \odot \langle \text{Nt : Network} \mid \text{soup} : (S_1 \leftarrow 10) \rangle \\ & \langle S_1 : \text{Server} \mid \text{queue} : \text{nil} \rangle \langle S_2 : \text{Server} \mid \text{queue} : (S_2 \leftarrow 5) \rangle \end{aligned}$$

is a context.

Definition 6 (R/A-matches). Given $[u]_A \in Can_{\Sigma, E/A}$, its R/A -matches are triples $([\mathbb{C}]_A, r, [\theta]_A)$, where if $r \in R$ is a rule

$$l : t(\vec{x}) \rightarrow t'(\vec{x}, \vec{y}) \text{ if } C(\vec{x}) \text{ with probability } \pi_r(\vec{x})$$

then $[\theta]_A \in \llbracket C \rrbracket$, that is $[\theta]_A$ satisfies condition C and $[u]_A = [\mathbb{C}(\odot \leftarrow \theta(t))]_A$, so $[u]_A$ is the same as θ applied to the term $t(\vec{x})$ and placed in the context.

Consider the canonical-term

$$\langle H : \text{Client} \mid \text{sent} : 3, \text{svc}_1 : 1, \text{svc}_2 : 0 \rangle \langle Nt : \text{Network} \mid \text{soup} : (S_1 \leftarrow 10) \rangle \\ \langle S_1 : \text{Server} \mid \text{queue} : \text{nil} \rangle \langle S_2 : \text{Server} \mid \text{queue} : (S_2 \leftarrow 5) \rangle$$

Looking at the code in Figure 1, one of the R/A -matches for the equivalence class of the above term is the triple $([\mathbb{C}]_A, r, [\theta]_A)$ such that

$$\mathbb{C} = \odot \langle Nt : \text{Network} \mid \text{soup} : (S_1 \leftarrow 10) \rangle \\ \langle S_1 : \text{Server} \mid \text{queue} : \text{nil} \rangle \langle S_2 : \text{Server} \mid \text{queue} : (S_2 \leftarrow 5) \rangle$$

and θ is such that

$$\theta(Cl) = H, \theta(N) = 3, \theta(m_1) = 1, \theta(m_2) = 0.$$

Definition 7 (E/A-canonical one-step \mathcal{R} -rewrite). An E/A-canonical one-step \mathcal{R} -rewrite is a labelled transition of the form,

$$[u]_A \xrightarrow{([\mathbb{C}]_A, r, [\theta]_A, [\rho]_A)} [v]_A$$

where

1. $[u]_A, [v]_A \in \text{Can}_{\Sigma, E/A}$
2. $([\mathbb{C}]_A, r, [\theta]_A)$ is an R/A -match of $[u]_A$
3. $[\rho]_A \in \text{CanGSubst}_{E/A}(\vec{y})$
4. $[v]_A = [\mathbb{C}(\odot \leftarrow t'(\theta(\vec{x}), \rho(\vec{y})))]_A$, where $\{\theta, \rho\}|_{\vec{x}} = \theta$ and $\{\theta, \rho\}|_{\vec{y}} = \rho$.

We associate the probability $\pi_r(\alpha_r(\rho))$ with this transition. We can now see why the \mathcal{F}_r cover α_r was needed. The nondeterminism associated with the choice of the R/A -match must be removed in order to associate a probability space over the space of computations (which are infinite sequences of canonical one step \mathcal{R} -rewrites). The nondeterminism is removed by what is called an *adversary* of the system, which defines a probability distribution over the set of R/A -matches. In [9] a probability space is associated over the set of computation paths. To do this, an adversary for the system is fixed. We have also shown in [9] that probabilistic rewrite theories have great expressive power. They can express various known models of probabilistic systems like Continuous Time Markov Chains [8], Markov Decision Processes [10] and even Generalized Semi Markov Processes [7]. We also propose a temporal logic, to express properties of interest in probabilistic systems. The details can be found in [9].

Probabilistic rewrite theories can be used to model probabilistic actor systems [1]. Actors, which are inherently asynchronous, can be modelled naturally using object based rewriting. In probabilistic actor systems we may be interested in modelling message delay distributions among other probabilistic entities. However because time acts as a global synchronization parameter the natural encoding using objects, computing by their own rewrite rules is insufficient. The technique of *delayed* messages helps us to correctly encode

time in actors. Actor failures and message drops can also be encoded. Due to space constraints we do not indicate our encoding in this paper. The file at <http://maude.cs.uiuc.edu/pmaude/at.maude> presents our technique.

A special subclass of **PRwTh**, called *finitary probabilistic rewrite theories*, while fairly general, are easier to implement. We describe them below.

4 Finitary Probabilistic Rewrite Theories

Observe that there are two kinds of nondeterministic choice involved in rewriting. First, the selection of the rule and second, the exact substitution-context pair. Instead of having to think of nondeterminism from both these sources, it is easier to think in terms of rewrite rules with same left-hand side term as representing the computation of some part of the system, say an object, and thus representing one nondeterministic choice. Of course the substitution and context also have to be chosen to fix a nondeterministic choice. After nondeterministically selecting a rewrite rule, instantiated with a given substitution in a given context, different probabilistic choices arise for different right-hand sides of rules having the same left-hand side as that of the chosen rule, and which can apply in the chosen context with the chosen substitution. To assign probabilities, we assign *rate* functions to rules with the same left-hand side and normalize them. The rates, which depend on the chosen substitution, correspond to the *frequency* with which the RHS's are selected. Moreover, not having new variables on the right-hand sides of rules makes the implementation much simpler. Such theories are called *finitary probabilistic rewrite theories*. We define them formally below.

Definition 8 (Finitary probabilistic rewrite theory). A *finitary probabilistic rewrite theory* is a 4-tuple $\mathcal{R}_f = (\Sigma, E \cup A, R, \gamma)$, with $(\Sigma, E \cup A, R)$ a rewrite theory and $\gamma : R \rightarrow T_{\Sigma, E/A, PosRat}(X)$ a function associating to each rewrite rule in R a term $\gamma(r) \in T_{\Sigma, E/A, PosRat}(X)$, with some variables from the set X , and of sort $PosRat$, where $PosRat$ is a sort in $(\Sigma, E \cup A)$ corresponding to the positive rationals. The term $\gamma(r)$ represents the *rate* function associated with rule $r \in R$. If $l : t(\vec{x}) \rightarrow t'(\vec{x})$ if $C(\vec{x})$ is a rule in R involving variables \vec{x} , then γ maps the rule to a term of the form $\gamma_r(\vec{x})$ possibly involving some of the variables in \vec{x} . We then use the notation

$$l : t(\vec{x}) \rightarrow t'(\vec{x}) \text{ if } C(\vec{x}) [\text{ rate } \gamma_r(\vec{x})]$$

for the γ -annotated rule. Notice that t' does not have any new variables. Thus, all variables in t' are also variables in t . Furthermore, we require that all rules labelled by l have the *same* left-hand side and are of the form

$$\begin{aligned} l : t &\rightarrow t'_1 \text{ if } C_1 [\text{ rate } \gamma_{r_1}(\vec{x})] \\ &\dots \\ l : t &\rightarrow t'_n \text{ if } C_n [\text{ rate } \gamma_{r_n}(\vec{x})] \end{aligned} \tag{1}$$

where

1. $\vec{x} = fvars(t) \supseteq \bigcup_{1 \leq i \leq n} fvars(t'_i) \cup fvars(C_i)$, that is the terms t'_i and the conditions C_i do not have any variables other than \vec{x} , the set of variables in t .
2. C_i is of the form $(\bigwedge_j u_{ij} = v_{ij}) \wedge (\bigwedge_k w_{ik} : s_{ik})$, that is, condition C_i is a conjunction of equations and memberships.¹

We denote the class of finitary probabilistic rewrite theories by **FPRTh**.

4.1 Semantics of Finitary Probabilistic Rewrite Theories

Given a finitary probabilistic rewrite theory $\mathcal{R}_f = (\Sigma, E \cup A, R, \gamma)$, we can express it as a probabilistic rewrite theory \mathcal{R}_f^\bullet , by defining a map $F_R : \mathcal{R}_f \mapsto \mathcal{R}_f^\bullet$, with $\mathcal{R}_f^\bullet = (\Sigma^\bullet, E^\bullet \cup A, R^\bullet, \pi^\bullet)$ and $(\Sigma, E \cup A) \subseteq (\Sigma^\bullet, E^\bullet \cup A)$, in the following way. We encode each group of rules in R with label l of the form 1 above by a single probabilistic rewrite rule²

$$t(\vec{x}) \rightarrow proj(i, (t'_1(\vec{x}), \dots, t'_n(\vec{x}))) \text{ if } \tilde{C}_1(\vec{x}) \text{ or } \dots \text{ or } \tilde{C}_n(\vec{x}) = true \\ \text{with probability } \pi_r(\vec{x})$$

in R^\bullet . Corresponding to each such rule, we add to Σ^\bullet the sort $[1 : n]$, with constants $1, \dots, n : [1 : n]$, and the projection operator $proj : [1 : n] k \dots k \rightarrow k$. We also add to E^\bullet the equations $proj(i, t_1, \dots, t_n) = t_i$ for each $i \in \{1, \dots, n\}$. Note that the only new variable on the righthand side is i , and therefore $CanGSubst_{E/A}(i) \cong \{1, \dots, n\}$. We consider the σ -algebra $\mathcal{P}(\{1, \dots, n\})$ on $\{1, \dots, n\}$. Then π_r is a function

$$\pi_r : \llbracket C \rrbracket \rightarrow PFun(\{1, \dots, n\}, \mathcal{P}(\{1, \dots, n\}))$$

defined as follows. If θ is such that $\tilde{C}_1(\theta(\vec{x})) \text{ or } \dots \text{ or } \tilde{C}_n(\theta(\vec{x})) = true$, then $\pi_\theta = \pi_r(\theta)$ defined as

$$\pi_\theta(\{i\}) = \frac{? \gamma_{r_i}(\theta(\vec{x}))}{? \gamma_{r_1}(\theta(\vec{x})) + ? \gamma_{r_2}(\theta(\vec{x})) + \dots + ? \gamma_{r_n}(\theta(\vec{x}))}$$

where, if $\tilde{C}_i(\theta(\vec{x})) = true$, then $? \gamma_{r_i}(\theta(\vec{x})) = \gamma_{r_i}(\theta(\vec{x}))$ and $? \gamma_{r_i}(\theta(\vec{x})) = 0$ otherwise. The semantics of \mathcal{R}_f computations is now defined in terms of its associated theory \mathcal{R}_f^\bullet in the standard way, by choosing the singleton \mathcal{F} -cover $\alpha_r : \{1, \dots, n\} \rightarrow \mathcal{P}(\{1, \dots, n\})$ mapping each i to $\{i\}$.

¹ The requirement $fvars(C_i) \subseteq fvars(t)$ can be relaxed by allowing new variables in C_i to be introduced in “matching equations” in the sense of [4]. Then these new variables can also appear in t'_i .

² By the assumption that $(\Sigma, E \cup A)$ is confluent, sort-decreasing, and terminating modulo A , and by a metatheorem of Bergstra and Tucker, any condition C of the form $(\bigwedge_i u_i = v_i \wedge \bigwedge_j w_j : s_j)$ can be replaced in an appropriate protecting enrichment $(\tilde{\Sigma}, \tilde{E} \cup A)$ of $(\Sigma, E \cup A)$ by a semantically equivalent Boolean condition $\tilde{C} = true$.

5 The PMaude Tool

We have developed an interpreter called **PMaude**, which provides a framework for specification and execution of finitary probabilistic rewrite theories. The **PMaude** interpreter has been built on top of Maude 2.0 [4,3] using the Full-Maude library [6]. We describe below how a finitary probabilistic rewrite theory is specified in our implemented framework and discuss some of the implementation details.

Consider a finitary probabilistic rewrite theory with k distinct rewrite labels and with n_i rewrite rules for the i^{th} distinct label, for $i = 1, 2, \dots, k$.

$$\begin{aligned} l_1 : t_1 &\rightarrow t'_{11} \quad \text{if } C_{11} [\text{rate } \gamma_{11}(\vec{x})] \\ &\dots \\ l_1 : t_1 &\rightarrow t'_{1n_1} \quad \text{if } C_{1n_1} [\text{rate } \gamma_{1n_1}(\vec{x})] \\ &\dots \\ l_k : t_k &\rightarrow t'_{k1} \quad \text{if } C_{k1} [\text{rate } \gamma_{k1}(\vec{x})] \\ &\dots \\ l_k : t_k &\rightarrow t'_{kn_k} \quad \text{if } C_{kn_k} [\text{rate } \gamma_{kn_k}(\vec{x})] \end{aligned}$$

At one level we want all rewrite rules in the specification to have *distinct* labels, so that we have low level control over these rules, while at the conceptual level, groups of rules must have the same label. We achieve this by giving two labels: one, common to a group and corresponding to the group's label l at the beginning, and another, unique for each rule, at the end. The above finitary probabilistic rewrite theory can be specified as follows in **PMaude**.

```
pmod FINITARY-EXAMPLE is
  cprl [ $l_1$ ] :  $t_1 \Rightarrow t'_{11}$  if  $C_{11}$  [rate  $\gamma_{11}(x_1, \dots)$ ] [metadata “ $l_{11} \dots$ ”] .
  ...
  cprl [ $l_1$ ] :  $t_1 \Rightarrow t'_{1n_1}$  if  $C_{1n_1}$  [rate  $\gamma_{1n_1}(x_1, \dots)$ ] [metadata “ $l_{1n_1} \dots$ ”] .
  ...
  cprl [ $l_k$ ] :  $t_k \Rightarrow t'_{k1}$  if  $C_{k1}$  [rate  $\gamma_{k1}(x_1, \dots)$ ] [metadata “ $l_{k1} \dots$ ”] .
  ...
  cprl [ $l_k$ ] :  $t_k \Rightarrow t'_{kn_k}$  if  $C_{kn_k}$  [rate  $\gamma_{kn_k}(x_1, \dots)$ ] [metadata “ $l_{kn_k} \dots$ ”] .
endpm
```

User input and output are supported as in Full Maude using the **LOOP-MODE** module. **PMaude** extends the Full Maude functions for parsing modules and any terms entered later by the user for rewriting purposes. Currently **PMaude** supports four user commands. Two of these are low level commands used to change seeds of pseudo-random generators. We shall not describe the implementation of those two commands here. The other two commands are rewrite commands. Their syntax is as follows:

```
(prew  $t .$ )
(prew-[n]  $t .$ )
```

The default module M in which these commands are interpreted is the last read probabilistic module. The *prew* command is an instruction to the interpreter

to probabilistically rewrite the term t in the default module M , till no further rewrites are possible. Notice that this command may fail to terminate. The *prew-[n]* command takes a natural number n specifying the maximum number of probabilistic rewrites to perform on the term t . This command always terminates in at most n steps of rewriting. Both commands report the final term (if *prew* terminates).

The implementation of these commands is as follows. When the interpreter is given one of these commands, all possible one-step rewrites for t in the default module M are computed. Out of all possible groups l_1, l_2, \dots, l_k in which *some* rewrite rule applies, one is chosen, *uniformly at random*. For the chosen group l_i , all the rewrite rules $l_{i1}, l_{i2}, \dots, l_{in_i}$ associated with l_i , are guaranteed to have the same left-hand side $t_i(x_1, x_2, \dots)$. From all possible canonical substitution, context pairs $([\theta]_A, [C]_A)$ for the variables x_j , representing successful matches of $t_i(x_1, x_2, \dots)$ with the given term t , that is, matches that also satisfy one of the conditions C_{ij} , one of the matches is chosen *uniformly at random*. The two steps above also define the exact adversary we associate to a given finitary probabilistic rewrite theory in our implementation. If there are m groups, l_{i1}, \dots, l_{im} , in which *some* rule applies and v_j matches in total for group l_{ij} then the adversary chooses a match in group l_{ij} with probability $\frac{1}{mv_j}$. To choose the exact rewrite rule l_{ij} to apply, use of the rate functions is made. The values of the various rates γ_{ip} are calculated for those rules l_{ip} such that $[\theta]_A$ satisfies the condition of the rule l_{ip} . Then these rates are normalized and the choice of the rule l_{ij} is made *probabilistically*, based on the calculated rates. This rewrite rule is then applied to the term t , in the chosen context with the chosen substitution. If the interpreter finds no successful matches for a given term, or if it has completed the maximum number of rewrites specified, it immediately reports that term as the answer. Since rates can depend on the substitution, this allows users to specify systems where probabilities are determined by the state.

PMaude can be used as a simulator for finitary probabilistic rewrite theories. The programmer must supply the system specification as a **PMaude** module and a start term to rewrite. To obtain different results the seeds for the random number generators must be changed at each invocation. This can be done by using a scripting language to call the interpreter repeatedly but with different seeds before each execution.

We have specified the client-server example discussed in Section 3 in **PMaude** with the following parameters: The client only sends two kinds of packets, loads 5 and 10, with equal probability. The request messages for S_1, S_2 are dropped with probabilities $2/7$ and $1/6$ respectively, while acknowledgement messages for S_1, S_2 are dropped with probabilities $1/6, 1/11$ respectively. We also chose S_1 to drop processing of a request with probability $1/7$ as opposed to $3/23$ when its load was at least 100, while for S_2 the load limit was 40 but the probabilities of dropping requests were $1/7$ and $3/19$. We performed some simulations and, after a number of runs, we computed the ratio $(svc_1 + svc_2)/sent$ as a measure of the quality of service for the client. Simulations showed that among static policies, namely those where the client did not adapt sending probabilities

on the fly, that of *sending twice as often* to server S_2 than to S_1 was better than most others.

The complete code for the **PMaude** interpreter, as well as several other example files, can be found at <http://maude.cs.uiuc.edu/pmaude/pmaude.html>.

6 Conclusions and Future Work

Probabilistic rewrite theories provide a general semantic framework supporting high level probabilistic specification of systems; in fact, we have shown how various well known probabilistic models can be expressed in our framework [9]. The present work shows how our framework applies to concurrent object based systems. For a fairly general subclass, namely finitary probabilistic rewrite theories, we have implemented a simulator **PMaude** and have exercised it on some simple examples. We are currently carrying out more case studies. We have also identified several aspects of the theory and the **PMaude** tool that need further development.

On the more theoretical side, we feel research is needed in three areas. First, it is important to develop a general model of probabilistic systems with *concurrent* probabilistic actions, as opposed to the current *interleaving* semantics. Second, deductive and analytic methods for property verification of probabilistic systems, based on our current framework is an important open problem. Algorithms to translate appropriate subclasses to appropriate representations enabling use of existing model checkers, should be developed and implemented.

Third, we think that allowing the probability function π_r to depend not only on the substitution, but also on the context would give us more modelling power. Specifically, it would enable us to represent applications where the probability distributions of certain variables, such as message delays, depend on functions of the entire state of the system, for example, on the congestion in the network. Such situations can also be modelled in our current framework but at the expense of using rewrite rules at the *top-level*, whose substitutions capture global system parameters. Such rules can modify the entire system at once as opposed to just modifying local fragments, but at the cost of violating the modularity principle of concurrent objects or actors.

On the implementation side, an extension of the **PMaude** framework to enable specification of more general classes of probabilistic rewrite theories and adversaries is highly desirable. This will allow the generation of simulation traces for the system under consideration and can be used as a tool to implement the model independent Monte-Carlo simulation and acceptance sampling methods for probabilistic validation of properties [14]. As an application of our theory, we believe that it will be fruitful to model networked embedded systems, where apart from time, there are other continuous state variables, such as battery power or temperature, whose behavior may be stochastic. Moreover, the properties of interest are often a statistical aggregation of many observations.

7 Acknowledgement

The work is supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586 and the DARPA IXO NEST Program, contract number F33615-01-C-1907) and the ONR Grant N00014-02-1-0715. We would like to thank Wooyoung Kim for reading a previous version of this paper and giving us valuable feedback and Joost-Pieter Katoen for very helpful discussions and pointers to references.

References

1. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.
2. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1–2):35–132, 2000.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Towards maude 2.0. In K. Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier, 2001.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual, Version 1.0*, june 2003. <http://maude.cs.uiuc.edu/manual/maude-manual.pdf>.
6. F. Durán and J. Meseguer. Parameterized theories and views in full maude 2.0. In K. Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier, 2001.
7. P. Glynn. The role of generalized semi-Markov processes in simulation output analysis, 1983.
8. J.-P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and symbolic CTMC model checking. *Lecture Notes in Computer Science*, 2165, 2001.
9. N. Kumar, K. Sen, J. Meseguer, and G. Agha. Probabilistic rewrite theories: Unifying models, logics and tools. Technical Report UIUCDCS-R-2003-2347, University of Illinois at Urbana-Champaign, May 2003.
10. M. Z. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker, 2002.
11. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
12. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.
13. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT’97*, pages 18–61. Springer LNCS 1376, 1998.
14. H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 223–235, Copenhagen, Denmark, July 2002. Springer.

Appendix AE:

An Executable Specification of Asynchronous Pi-Calculus Semantics and May Testing in Maude 2.0

Prasanna Thati Koushik Sen

*Department of Computer Science
University of Illinois at Urbana-Champaign
{thati,ksen}@cs.uiuc.edu*

Narciso Martí-Oliet

*Dpto. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
narciso@sip.ucm.es*

Abstract

We describe an executable specification of the operational semantics of an asynchronous version of the π -calculus in Maude by means of conditional rewrite rules with rewrites in the conditions. We also present an executable specification of the may testing equivalence on non-recursive asynchronous π -calculus processes, using the Maude metalevel. Specifically, we describe our use of the `metaSearch` operation to both calculate the set of all finite traces of a non-recursive process, and to compare the trace sets of two processes according to a preorder relation that characterizes may testing in asynchronous π -calculus. Thus, in both the specification of the operational semantics and the may testing, we make heavy use of new features introduced in version 2.0 of the Maude language and system.

Key words: π -calculus, asynchrony, may testing, traces, Maude.

1 Introduction

Since its introduction in the seminal paper [11] by Milner, Parrow, and Walker, the π -calculus has become one of the most studied calculus for name-based mobility of processes, where processes are able to exchange names over channels so that the communication topology can change during the computation. The operational semantics of the π -calculus has been defined for several different versions of the calculus following two main styles. The first is the labelled transition system style according to the SOS approach introduced by Plotkin

©2003 Published by Elsevier Science B. V.

[13]. The second is the reduction style, where first an equivalence is imposed on syntactic processes (typically to make syntax more abstract with respect to properties of associativity and/or commutativity of some operators), and then some reduction or rewrite rules express how the computation proceeds by communication between processes.

The first specification of the π -calculus operational semantics in rewriting logic was developed by Viry in [19], in a reduction style making use of de Bruijn indexes, explicit substitutions, and reduction strategies in Elan [6]. This presentation was later improved by Stehr [14] by making use of a generic calculus for explicit substitutions, known as *CINNI*, which combines the best of the approaches based on standard variables and de Bruijn indices, and that has been implemented in Maude.

Our work took the work described above as a starting point, together with recent work by Verdejo and Martí-Oliet [18] showing how to use the new features of Maude 2.0 in the implementation of a semantics in the labelled transition system style for CCS. This work makes essential use of conditional rewrite rules with rewrites in the conditions, so that an inference rule in the labelled transition system of the form

$$\frac{P_1 \rightarrow Q_1 \quad \dots \quad P_n \rightarrow Q_n}{P_0 \rightarrow Q_0}$$

becomes a *conditional* rewrite rule of the form

$$P_0 \longrightarrow Q_0 \quad \text{if} \quad P_1 \longrightarrow Q_1 \wedge \dots \wedge P_n \longrightarrow Q_n,$$

where the condition includes rewrites. These rules are executable in version 2.0 of the Maude language and system [7]. However, this is not enough, because it is necessary to have some control on the application of rules. Typically, rewrite rules can be applied anywhere in a term, while the transitions in the operational semantics for CCS or the π -calculus in the SOS style only take place at the top. The new **frozen** attribute available in Maude 2.0 makes this possible, because the declaration of an operator as frozen forbids rewriting its arguments, thus providing another way of controlling the rewriting process. Rewrite conditions when applying conditional rules are solved by means of an implicit *search* process, which is also available to the user both at the command level and at the metalevel. The **search** command looks for all the rewrites of a given term that match a given pattern satisfying some condition. Search is reified at the metalevel as an operation **metaSearch**.

In this way, our first contribution is a fully executable specification of an operational semantics in the labelled transition system style for an asynchronous version of the π -calculus (the semantics for the synchronous case is obtained as a simple modification). This specification uses conditional rewrite rules with rewrites in conditions and the CINNI calculus [14] for managing names and bindings in the π -calculus. However, these two ingredients are not enough to obtain a fully executable specification. A central problem to overcome is that the transitions of a term can be *infinitely branching*. For instance,

the term $x(y).P$ can evolve via an input action to one of an infinite family of terms depending on the name received in the input at channel x . Our solution is to define the transitions of a process relative to an execution environment. The environment is represented abstractly as a set of free (global) names that the environment may use while interacting with the process, and transitions are modelled as rewrite rules over a pair consisting of a set of environment names together with a process.

Our next contribution is to implement the verification of the *may-testing preorder* [12,3,5] between finitary (non-recursive) asynchronous π -calculus processes, using again ideas from [18] to calculate the set of all finite traces of a process. May testing is a specific instance of the notion of behavioral equivalence on π -calculus processes; in may testing, two processes are said to be equivalent if they have the same success properties in all experiments. An experiment consists of an observing process that runs in parallel and interacts with the process being tested, and success is defined as the observer signalling a special event. Viewing the occurrence of an event as something bad happening, may testing can be used to reason about safety properties [4].

Since the definition of may testing involves a universal quantification over all observers, it is difficult to establish process equivalences directly from the definition. As a solution, alternate characterizations of the equivalence that do not resort to quantification over observers have been found. It is known that the trace semantics is an alternate characterization of may testing in (synchronous) π -calculus [3], while a variant of the trace semantics has been shown to characterize may testing in an asynchronous setting [5]. Specifically, in both these cases, comparing two processes according to the may-testing preorder amounts to comparing the set of all finite traces they exhibit. We have implemented for finite asynchronous processes, the comparison of trace sets proposed in [5]. We stress that our choice of specifying an asynchronous version rather than the synchronous π -calculus, is because the characterization of may testing for the asynchronous case is more interesting and difficult. The synchronous version can be specified in an executable way using similar but simpler techniques.

Our first step in obtaining an executable specification of may testing is to obtain the set of all finite traces of a given process. This is done at the Maude metalevel by using the `metaSearch` operation to collect all results of rewriting a given term. The second step is to specify a preorder relation between traces that characterizes may testing. We have represented the trace preorder relation as a rewriting relation, i.e. the rules of inference that define the trace preorder are again modeled as conditional rewrite rules. The final step is to check if two processes are related by the may preorder, i.e. whether a statement of the form $P \sqsubseteq Q$ is true or not. This step involves computing the closure of a trace under the trace-preorder relation, again by means of the `metaSearch` operation. Thus, our work demonstrates the utility of the new metalevel facilities available in Maude 2.0.

The structure of the paper follows the steps in the description above. Section 2 describes the syntax of the asynchronous version of the π -calculus that we consider, together with the corresponding CINNI operations we use. Section 3 describes the operational semantics specified by means of conditional rewrite rules. Sections 4 and 5 define traces and the preorder on traces, respectively. Finally, Section 6 contains the specification of the may testing on processes as described above. Section 7 concludes the paper along with a brief discussion of future work.

Although this paper includes some information on the π -calculus and may testing to make it as self contained as possible, we refer the reader to the papers [5,3,11] for complete details on these subjects. In the same way, the interested reader can find a detailed explanation about the new features of Maude 2.0 in [7], and about their use in the implementation of operational semantics in the companion paper [18].

2 Asynchronous π -Calculus Syntax

The following is a brief and informal review of a version of asynchronous π -calculus that is equipped with a conditional construct for matching names. An infinite set of channel names is assumed, and u, v, w, x, y, z, \dots are assumed to range over it. The set of processes, ranged over by P, Q, R , is defined by the following grammar:

$$P := \bar{x}y \mid \sum_{i \in I} \alpha_i.P_i \mid P_1|P_2 \mid (\nu x)P \mid [x = y](P_1, P_2) \mid !P$$

where α can be $x(y)$ or τ .

The output term $\bar{x}y$ denotes an asynchronous message with target x and content y . The summation $\sum_{i \in I} \alpha_i.P_i$ non-deterministically chooses an α_i , and if $\alpha_i = \tau$ it evolves internally to P_i , and if $\alpha_i = x(y)$ it receives an arbitrary name z at channel x and then behaves like $P\{z/y\}$. The process $P\{z/y\}$ is the result of the substitution of free occurrences of y in P by z , with the usual renaming of bound names to avoid accidental captures (thus substitution is defined only modulo α -equivalence). The argument y in $x(y).P$ binds all free occurrences of y in P . The composition $P_1|P_2$ consists of P_1 and P_2 acting in parallel. The components can act independently, and also interact with each other. The restriction $(\nu x)P$ behaves like P except that it can not exchange messages targeted to x , with its environment. The restriction binds free occurrences of x in P . The conditional $[x = y](P_1, P_2)$ behaves like P_1 if x and y are identical, and like P_2 otherwise. The replication $!P$ provides an infinite number of copies of P . The functions for free names $fn(\cdot)$, bound names $bn(\cdot)$ and names $n(\cdot)$, of a process, are defined as expected.

In the Maude specification for the π -calculus syntax that follows, the sort `Chan` is used to represent channel names and each of the non-constant syntax constructors is declared as `frozen`, so that the corresponding arguments

cannot be rewritten by rules; this will be justified at the end of Section 3.

```

sort Chan .
sorts Guard GuardedTrm SumTrm Trm .
subsort GuardedTrm < SumTrm .
subsort SumTrm < Trm .

op _(_ ) : Chan Qid -> Guard .
op tau : -> Guard .
op nil : -> Trm .
op _<_> : Chan Chan -> Trm [frozen] .
op _.._ : Guard Trm -> GuardedTrm [frozen] .
op _+_ : SumTrm SumTrm -> SumTrm [frozen assoc comm] .
op _|_ : Trm Trm -> Trm [frozen assoc comm] .
op new[_]_ : Qid Trm -> Trm [frozen] .
op if_=then_else_fi : Chan Chan Trm Trm -> Trm [frozen] .
op !_ : Trm -> Trm [frozen] .

```

Note that the syntactic form $\sum_{i \in I} \alpha_i.P_i$ has been split into three cases:

- (i) `nil` represents the case where $I = \emptyset$,
- (ii) a term of sort `GuardedTrm` represents the case where $I = \{1\}$, and
- (iii) a term of sort `SumTrm` represents the case where $I = [1..n]$ for $n > 1$. Since the constructor `_+_` is associative and the sort `GuardedTrm` is a subsort of `SumTrm`, we can represent a finite sum $\sum_{i \in I} \alpha_i.P_i$ as $(\dots (\alpha_1.P_1 + \alpha_2.P_2) + \dots \alpha_n.P_n)$.

To represent substitution on π -calculus processes (and traces, see Section 4) at the language level we use CINNI as a calculus for explicit substitutions [14]. This gives a first-order representation of terms with bindings and capture-free substitutions, instead of going to the metalevel to handle names and bindings. The main idea in such a representation is to keep the bound names inside the binders as it is, but to replace its use by the name followed by an index which is a count of the number of binders with the same name it jumps before it reaches the place of use. Following this idea, we define terms of sort `Chan` as indexed names as follows.

```

sort Chan .
op _{_} : Qid Nat -> Chan [prec 1] .

```

We introduce a sort of substitutions `Subst` together with the following operations:

```

op [_:=_] : Qid Chan -> Subst .
op [shiftup_] : Qid -> Subst .
op [shiftdown_] : Qid -> Subst .
op [lift_] : Qid Subst -> Subst .

```

The first two substitutions are basic substitutions representing *simple* and *shiftup* substitutions; the third substitution is a special case of *simple* substitution; the last one represents complex substitution where a substitution can be lifted using the operator `lift`. The intuitive meaning of these operations

[a := x]	[shiftup a]	[shiftdown a]	[lift a S]
a{0} ↪ x	a{0} ↪ a{1}	a{0} ↪ a{0}	a{0} ↪ [shiftup a] (S a{0})
a{1} ↪ a{0}	a{1} ↪ a{2}	a{1} ↪ a{0}	a{1} ↪ [shiftup a] (S a{1})
...
a{n+1} ↪ a{n}	a{n} ↪ a{n+1}	a{n+1} ↪ a{n}	a{n} ↪ [shiftup a] (S a{n})
b{m} ↪ b{m}	b{m} ↪ b{m}	b{m} ↪ b{m}	b{m} ↪ [shiftup a] (S b{m})

Table 1
The CINNI operations.

is described in Table 1 (see [14] for more details). Using these, explicit substitutions for π -calculus processes are defined equationally. Some interesting equations are the following:

$$\begin{aligned} \text{eq } & S(P + Q) = (S P) + (S Q) . \\ \text{eq } & S(CX(Y) . P) = (S CX)(Y) . ([lift Y S] P) . \\ \text{eq } & S(\text{new } [X] P) = \text{new } [X] ([lift X S] P) . \end{aligned}$$

3 Operational Semantics

A labelled transition system (see Table 2) is used to give an operational semantics for the calculus as in [5]. The transition system is defined modulo α -equivalence on processes in that α -equivalent processes have the same transitions. The rules *COM*, *CLOSE*, and *PAR* have symmetric versions that are not shown in the table.

Transition labels, which are also called *actions*, can be of five forms: τ (a silent action), $\bar{x}y$ (free output of a message with target x and content y), $\bar{x}(y)$ (bound output), xy (free input of a message), and $x(y)$ (bound input). The functions $fn(.)$, $bn(.)$ and $n(.)$ are defined on actions as expected. The set of all visible (non- τ) actions is denoted by \mathcal{L} , and α is assumed to range over \mathcal{L} . As a uniform notation for free and bound actions the following notational convention is adopted: $(\emptyset)\bar{x}y = \bar{x}y$, $(\{y\})\bar{x}y = \bar{x}(y)$, and similarly for input actions. The variable \hat{z} is assumed to range over $\{\emptyset, \{z\}\}$. The term $(\nu z)P$ is $(\nu z)P$ if $\hat{z} = \{z\}$, and P otherwise.

We define the sort **Action** and the corresponding operations as follows:

```
sorts Action ActionType .
ops i o : -> ActionType .
op f : ActionType Chan Chan -> Action .
op b : ActionType Chan Qid -> Action .
op tauAct : -> Action .
```

The operators **f** and **b** are used to construct free and bound actions respectively. Name substitution on actions is defined equationally as expected.

The inference rules in Table 2 are modelled as conditional rewrite rules

$INP: \sum_{i \in I} \alpha_i.P_i \xrightarrow{x_j z} P_j\{z/y\} \ j \in I, \alpha_j = x_j(y)$	$OUT: \bar{x}y \xrightarrow{\bar{x}y} 0$
$TAU: \sum_{i \in I} \alpha_i.P_i \xrightarrow{\tau} P_j \ j \in I, \alpha_j = \tau$	$BINP: \frac{P \xrightarrow{x y} P'}{P \xrightarrow{x(y)} P'} y \notin fn(P)$
$PAR: \frac{P_1 \xrightarrow{\alpha} P'_1}{P_1 P_2 \xrightarrow{\alpha} P'_1 P_2} bn(\alpha) \cap fn(P_2) = \emptyset$	$COM: \frac{P_1 \xrightarrow{\bar{x}y} P'_1 \quad P_2 \xrightarrow{x y} P'_2}{P_1 P_2 \xrightarrow{\tau} P'_1 P'_2}$
$RES: \frac{P \xrightarrow{\alpha} P'}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'} y \notin n(\alpha)$	$OPEN: \frac{P \xrightarrow{\bar{x}y} P'}{(\nu y)P \xrightarrow{\bar{x}(y)} P'} x \neq y$
$CLOSE: \frac{P_1 \xrightarrow{\bar{x}(y)} P'_1 \quad P_2 \xrightarrow{x y} P'_2}{P_1 P_2 \xrightarrow{\tau} (\nu y)(P'_1 P'_2)} y \notin fn(P_2)$	$REP: \frac{P !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$
$IF: \frac{P \xrightarrow{\alpha} P'}{[x = x](P, Q) \xrightarrow{\alpha} P'}$	$ELSE: \frac{Q \xrightarrow{\alpha} Q'}{[x = y](P, Q) \xrightarrow{\alpha} Q'} x \neq y$

Table 2
A labelled transition system for asynchronous π -calculus.

with the premises as conditions of the rule.¹ Since rewrites do not have labels unlike the labelled transitions, we make the label a part of the resulting term; thus rewrites corresponding to transitions in the operational semantics are of the form $P \Rightarrow \{\alpha\}Q$.

Because of the INP and $OPEN$ rules, the transitions of a term can be infinitely branching. Specifically, in case of the INP rule there is one branch for every possible name that can be received in the input. In case of the $OPEN$ rule, there is one branch for every name that is chosen to denote the private channel that is being emitted (note that the transition rules are defined only modulo α -equivalence). To overcome this problem, we define transitions over pairs of the form $[CS] \ P$, where CS is a set of channel names containing all the names that the environment with which the process interacts, knows about. The set CS expands during bound input and output interactions when private names are exchanged between the process and its environment.

The infinite branching due to the INP rule is avoided by allowing only the names in the environment set CS to be received in free inputs. Since CS is assumed to contain all the free names in the environment, an input argument that is not in CS would be a private name of the environment. Now, since the identifier chosen to denote the fresh name is irrelevant, all bound input

¹ The symmetric versions missing in the table need not be implemented because the process constructors $_+_$ and $_|__$ have been declared as commutative.

transitions can be identified to a single input. With these simplifications, the number of input transitions of a term become finite. Similarly, in the *OPEN* rule, since the identifier chosen to denote the private name emitted is irrelevant, instances of the rule that differ only in the chosen name are not distinguished.

We discuss in detail the implementation of only a few of the inference rules; the reader is referred to the appendix for a complete list of all the rewrite rules for Table 2.

```
sorts EnvTrm TraceTrm .
subsort EnvTrm < TraceTrm .
op  []_ : Chanset Trm -> EnvTrm [frozen] .
op  {}_ : Action TraceTrm -> TraceTrm [frozen] .
```

Note that the two operators are also declared above with the `frozen` attribute, forbidding in this way rewriting of their arguments, as justified at the end of this section.

The following non-conditional rule is for free inputs.

```
r1 [Inp] : [CY CS] ((CX(X) . P) + SUM) =>
           {f(i,CX,CY)} ([CY CS] ([X := CY] P)) .
```

The next rule we consider is the one for bound inputs. Since the identifier chosen to denote the bound argument is irrelevant, we use the constant '`U`' for all bound inputs, and thus '`U{0}`' denotes the fresh channel received. Note that in contrast to the *BINP* rule of Table 2, we do not check if '`U{0}`' is in the free names of the process performing the input, and instead we shift up the channel indices appropriately, in both the set of environment names `CS` and the process `P` in the righthand side and condition of the rule. This is justified because the transition target is within the scope of the bound name in the input action. Note also that the channel `CX` in the action is not shifted down because it is out of the scope of the bound argument. The set of environment names is expanded by adding the received channel '`U{0}`' to it. Finally, we use a special constant `flag` of sort `Chan`, to ensure termination. We add an instance of `flag` to the environment set of the rewrite in condition, so that the *BINP* rule is not fired again while evaluating the condition. Without this check, we will have a non-terminating execution in which the *BINP* rule is repeatedly fired.

```
crl [BInp] : [CS] P => {b(i,CX,'U)} ['U{0} [shiftup 'U] CS] P1
    if (not flag in CS) /\
        CS1 := flag 'U{0} [shiftup 'U] CS /\
        [CS1] [shiftup 'U] P => {f(i,CX,'U{0})} [CS1] P1 .
```

The following rule treats the case of bound outputs.

```
crl [Open] : [CS] (new [X] P) => {[shiftdown X] b(o,CY,X)} [X{0} CS1] P1
    if CS1 := [shiftup X] CS /\
        [CS1] P => {f(o,CY,X{0})} [CS1] P1 /\
        X{0} /= CY .
```

Like in the case of bound inputs, we identify all bound outputs to a single

instance in which the identifier X that appears in the restriction is chosen as the bound argument name. Note that in both the righthand side of the rule and in the condition, the indices of the channels in CS are shifted up, because they are effectively moved across the restriction. Similarly, the channel indices in the action in the righthand side of the rule are shifted down since the action is now moved out of the restriction. Note also that the exported name is added to the set of environment names, because the environment that receives this exported name can use it in subsequent interactions.

The PAR inference rule is implemented by two rewrite rules, one for the case where the performed action is free, and the other where the action is bound. The rewrite rule for the latter case is discussed next, while the one for the former case is simpler and appears in the appendix.

```
var IO : ActionType
crl [Par] : [CS] (P | Q) =>
    {b(IO,CX,Y)} [Y{0}] ([shiftup Y] CS) (P1 | [shiftup Y] Q)
    if [CS] P => {b(IO,CX,Y)} ([CS1] P1) .
```

Note that the side condition of the PAR rule in Table 2, which avoids confusion of the emitted bound name with free names in Q , is achieved by shifting up channel indices in Q . This is justified because the righthand side of the rule is under the scope of the bound output action. Similarly, the channel indices in the environment are also shifted up. Further, the set of environment names is expanded by adding the exported channel $Y\{0\}$.

Finally, we consider the rewrite rule for $CLOSE$. The process P emits a bound name Y , which is received by process Q . Since the scope of Y after the transition includes Q , the rewrite involving Q in the second condition of the rule is carried out within the scope of the bound name that is emitted. This is achieved by adding the channel $Y\{0\}$ to the set of environment names and shifting up the channel indices in both CS and Q in the rewrite. Note that since the private name being exchanged is not emitted to the environment, we neither expand the set CS in the righthand side of the rule nor shift up the channel indices in it.

```
crl [Close] : [CS] (P | Q) => {tauAct} [CS] new [Y] (P1 | Q1)
    if [CS] P => {b(o,CX,Y)} [CS1] P1 ∧
        [Y{0}] [shiftup Y] CS [shiftup Y] Q =>
            {f(i,CX,Y{0})} [CS2] Q1 .
```

We conclude this section with the following note. The operator $\{\cdot\}_-$ is declared **frozen** because further rewrites of the process term encapsulated in a term of sort **TraceTrm** are useless. This is because all the conditions of the transition rules only involve one step rewrites (the righthand side of these rewrites can only match a term of sort **TraceTrm** with a single action prefix). Further note that, to prevent rewrites of a term to a non well-formed term, all the constructors for π -calculus terms (Section 2) have been declared **frozen**; in the absence of this declaration we would have for instance rewrites of the form $P | Q \Rightarrow \{A\}.P1 | Q$ to a non well-formed term.

4 Trace Semantics

The set \mathcal{L}^* is the set of *traces*. The functions $fn(\cdot)$, $bn(\cdot)$ and $n(\cdot)$ are extended to \mathcal{L}^* in the obvious way. The relation of α -equivalence on traces is defined as expected, and α -equivalent traces are not distinguished. The relation \Rightarrow denotes the reflexive transitive closure of $\xrightarrow{\tau}$, and $\xrightarrow{\beta}$ denotes $\Rightarrow \xrightarrow{\beta} \Rightarrow$. For $s = l.s'$, we inductively define $P \xrightarrow{s} P'$ as $P \xrightarrow{l} \xrightarrow{s'} P'$. We use $P \xrightarrow{s}$ as an abbreviation for $P \xrightarrow{s} P'$ for some P' . The set of traces that a process exhibits is then $\|P\| = \{s \mid P \xrightarrow{s}\}$.

In the implementation, we introduce a sort **Trace** as supersort of **Action** to specify traces.

```
subsort Action < Trace .
op epsilon : -> Trace .
op _.._ : Trace Trace -> Trace [assoc id: epsilon] .
op [] : Trace -> TTrace .
```

We define the operator `[]` to represent a complete trace. The motivation for doing so is to restrict the equations and rewrite rules defined over traces to operate only on a complete trace instead of a part of it. The following equation defines α -equivalence on traces. Note that in a trace `TR1.b(I0,CX,Y).TR2` the action `b(I0,CX,Y)` binds the identifier `Y` in `TR2`.

```
ceq [TR1 . b(I0,CX,Y) . TR2] =
      [TR1 . b(I0,CX,'U) . [Y := 'U{0}] [shiftup 'U] TR2]
      if Y /= 'U .
```

Because the operator `op {}_- : Action TraceTrm -> TraceTrm` is declared as **frozen**, a term of sort **EnvTrm** can rewrite only once, and so we cannot obtain the set of finite traces of a process by simply rewriting it multiple times in all possible ways. The problem is solved as in [18], by specifying the trace semantics using rules that generate the transitive closure of one step transitions as follows:

```
sort TTrm .
op [] : EnvTrm -> TTrm [frozen] .
var TT : TraceTrm .

crl [reflx] : [ P ] => {A} Q if P => {A} Q .
crl [trans] : [ P ] => {A} TT
      if P => {A} Q /\ [ Q ] => TT /\ [ Q ] /= TT .
```

We use the operator `[]` to prevent infinite loops while evaluating the conditions of the rules above. If this operator were not used, then the lefthand side of the rewrite in the condition would match the lefthand side of the rule itself, and so the rule itself could be used in order to solve its condition. This operator is also declared as **frozen** to prevent useless rewrites inside `[]`.

We can now use the **search** command of Maude 2.0 to find all possible traces of a process. The traces appear as prefix of the one-step successors of a **TTrm** of the form `[[CS] P]`. For instance, the set of all traces exhibited

by $[\text{mt}] \text{ new } [y] (x_0 < y_0 > \mid x_0(u) . \text{nil})$ (where mt denotes the empty channel set), can be obtained by using the following `search` command.

```
Maude> search [ [mt] new [y] (x{0} < y{0} > | x{0}(u) . nil) ] =>!
X:TraceTrm .
search in APITRACESET : [[mt]new[y](x{0} < y{0} > | x{0}(u) . nil)] =>!
X:TraceTrm .

Solution 1 (state 1)
states: 7  rewrites: 17344 in 110ms cpu (150ms real) (157672 rewrites/second)
X:TraceTrm --> {b(i, 'x{0}, 'u)}['u{0}]new[y](nil | 'x{0} < 'y{0} >)

Solution 2 (state 2)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {tauAct}[mt]new[y](nil | nil)

Solution 3 (state 3)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(o, 'x{0}, 'y)}['y{0}]nil | 'x{0}(u) . nil

Solution 4 (state 4)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(i, 'x{0}, 'u)}{b(o, 'x{0}, 'y)}['y{0} 'u{0}]nil | nil

Solution 5 (state 5)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(o, 'x{0}, 'y)}{b(i, 'x{0}, 'u)}['y{0} 'u{0}]nil | nil

Solution 6 (state 6)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(o, 'x{0}, 'y)}{f(i, 'x{0}, 'y{0})}['y{0}]nil | nil

No more solutions.
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
```

The command returns all `TraceTrm`s that can be reached from the given `TTrm`, and that are terminating (the ‘!’ in $=>!$ specifies that the target should be terminating). The required set of traces can be obtained by simply extracting from each solution $\{a_1\} \dots \{a_n\} T T$ the sequence $a_1 \dots a_n$ and removing all `tauActs` in it. Thus, we have obtained an executable specification of the trace semantics of asynchronous π -calculus.

5 A Trace Based Characterization of May Testing

The may-testing framework [12] is instantiated on asynchronous π -calculus as follows. Observers are processes that can emit a special message $\bar{\mu}\mu$. We say that an observer O accepts a trace s if $O \xrightarrow{\bar{s}.\bar{\mu}\mu}$, where \bar{s} is the trace obtained by complementing the actions in s , i.e. converting input actions to output actions and vice versa. The may preorder \sqsubseteq over processes is defined as: $P \sqsubseteq Q$ if for every observer O , $P|O \xrightarrow{\bar{\mu}\mu}$ implies $Q|O \xrightarrow{\bar{\mu}\mu}$. We say that P and Q are *may-*

<i>(Drop)</i>	$s_1.(\hat{y})s_2 \prec s_1.(\hat{y})xy.s_2$ if $(\hat{y})s_2 \neq \perp$
<i>(Delay)</i>	$s_1.(\hat{y})(\alpha.xy.s_2) \prec s_1.(\hat{y})xy.\alpha.s_2$ if $(\hat{y})(\alpha.xy.s_2) \neq \perp$
<i>(Annihilate)</i>	$s_1.(\hat{y})s_2 \prec s_1.(\hat{y})xy.\bar{x}y.s_2$ if $(\hat{y})s_2 \neq \perp$

Table 3
A preorder relation on traces.

equivalent, i.e. $P = Q$, if $P \sqsubseteq Q$ and $Q \sqsubseteq P$. The universal quantification on contexts in this definition makes it very hard to prove equalities directly from the definition, and makes mechanical checking impossible. To circumvent this problem, a trace based alternate characterization of the may equivalence is proposed in [5]. We now summarize this characterization and discuss our implementation of it.

The preorder \preceq on traces is defined as the reflexive transitive closure of the laws shown in Table 3, where the notation $(\hat{y})\cdot$ is extended to traces as follows.

$$(\hat{y})s = \begin{cases} s & \text{if } \hat{y} = \emptyset \text{ or } b \notin fn(s) \\ s_1.x(y).s_2 & \text{if } \hat{y} = \{y\} \text{ and there are } s_1, s_2, x \text{ such that} \\ & s = s_1.xy.s_2 \text{ and } y \notin n(s_1) \cup \{x\} \\ \perp & \text{otherwise} \end{cases}$$

For sets of traces R and S , we define $R \precsim S$, if for every $s \in S$ there is an $r \in R$ such that $r \preceq s$. The may preorder is then characterized in [5] as: $P \sqsubseteq Q$ if and only if $\|Q\| \precsim \|P\|$.

The main intuition behind the preorder \preceq is that if an observer accepts a trace s , then it also accepts any trace $r \preceq s$. The first two laws state that an observer cannot force inputs on the process being tested. Since outputs are asynchronous, the actions following an output in a trace exhibited by the observer need not causally depend on the output. Hence the observer's output can be delayed until a causally dependent action, or dropped if there are no such actions. The annihilation law states that an observer can consume its own outputs unless there are subsequent actions that depend on the output. The reader is referred to [5] for further details on this characterization.

We encode the trace preorder as rewrite rules on terms of the sort **TTrace** of complete traces; specifically, the relation $r \prec s$ if cond, is encoded as **s => r if cond**. The reason for this form of representation will be justified in Section 6. The function $(\{y\})\cdot$ on traces is defined equationally by the operation **bind**. The constant **bot** of sort **Trace** is used by the bind operation to signal error.

```
op bind : Qid Trace -> Trace .
op bot  : -> Trace .
var TR  : Trace .    var IO  : ActionType.
```

```

ceq TR . bot = bot if t /= epsilon .
ceq bot . TR = bot if t /= epsilon .

eq bind(X , epsilon) = epsilon .

eq bind(X , f(i,CX,CY) . TR ) = if CX /= X{0} then
    if CY == X{0} then ([shiftdown X] b(i, CX , X)) . TR
    else ([shiftdown X] f(i, CX , CY)) . bind(X , TR) fi
    else bot fi .

eq bind(X , b(IO,CX,Y) . TR) = if CX /= X{0} then
    if X /= Y then ([shiftdown X] b(i, CX , Y)) . bind(X , TR)
    else ([shiftdown X] b(IO, CX , Y)) . bind(X , swap(X,TR)) fi
    else bot fi .

```

The equation for the case where the second argument to **bind** begins with a free output is not shown as it is similar. Note that the channel indices in actions until the first occurrence of $X\{0\}$ as the argument of a free input are shifted down as these move out of the scope of the binder X . Further, when a bound action with X as the bound argument is encountered, the **swap** operation is applied to the remaining suffix of the trace. The swap operation simply changes the channel indices in the suffix so that the binding relation is unchanged even as the binder X is moved across the bound action. This is accomplished by simultaneously substituting $X\{0\}$ with $X\{1\}$, and $X\{1\}$ with $X\{0\}$. Finally, note that when $X\{0\}$ is encountered as the argument of a free input, the input is converted to a bound input. If $X\{0\}$ is first encountered at any other place, an error is signalled by returning the constant **bot**.

The encoding of the preorder relation on traces is now straightforward.

```

crl [Drop] : [ TR1 . b(i,CX,Y) . TR2 ] => [ TR1 . bind(Y , TR2) ]
    if bind(Y , TR2) /= bot .

rl  [Delay] : [ ( TR1 . f(i,CX,CY) . b(IO,CU,V) . TR2 ) ] =>
    [ ( TR1 . b(IO,CU,V) . ([shiftup V] f(i, CX , CY)) . TR2 ) ] .

crl [Delay] : [ ( TR1 . b(i,CX,Y) . f(IO,CU,CV) . TR2 ) ] =>
    [ ( TR1 . bind(Y , f(IO,CU,CV) . f(i,CX,Y{0}) . TR2) ) ]
    if bind(Y , f(IO,CU,CV) . f(i,CX,Y{0}) . TR2) /= bot .

crl [Annihilate] : [ ( TR1 . b(i,CX,Y) . f(o,CX,Y{0}) . TR2 ) ] =>
    [ TR1 . bind(Y , TR2) ]
    if bind(Y , TR2) /= bot .

```

Note that in the first **Delay** rule, the channel indices of the free input action are shifted up when it is delayed across a bound action, since it gets into the scope of the bound argument. Similarly, in the second **Delay** rule, when the bound input action is delayed across a free input/output action, the channel indices of the free action are shifted down by the **bind** operation. The other two subcases of the **Delay** rule, namely, where a free input is to be delayed across a free input or output, and where a bound input is to be delayed across a bound input or output, are not shown as they are similar.

Similarly, for `Annihilate`, the case where a free input is to be annihilated with a free output is not shown.

6 Verifying the May Preorder between Finite Processes

We now describe our implementation of verification of the may preorder between finite processes, i.e. processes without replication, by exploiting the trace-based characterization of may testing discussed in Section 5. The finiteness of a process P only implies that the length of traces in $\|P\|$ is bounded, but the number of traces in $\|P\|$ can be infinite (even modulo α -equivalence) because the *INP* rule is infinitely branching. To avoid the problem of having to compare infinite sets, we observe that

$$\|Q\| \precsim \|P\| \quad \text{if and only if} \quad \|Q\|_{fn(P,Q)} \precsim \|P\|_{fn(P,Q)},$$

where for a set of traces S and a set of names ρ we define $S_\rho = \{s \in S \mid fn(s) \subseteq \rho\}$. Now, since the traces in $\|P\|$ and $\|Q\|$ are finite in length, it follows that the sets of traces $\|P\|_{fn(P,Q)}$ and $\|Q\|_{fn(P,Q)}$ are finite modulo α -equivalence. In fact, the set of traces generated for $[[fn(P,Q)] P]$ by our implementation described in Section 3, contains exactly one representative from each α -equivalence class of $\|P\|_{fn(P,Q)}$.

Given processes P and Q , we generate the set of all traces (modulo α -equivalence) of $[[fn(P,Q)] P]$ and $[[fn(P,Q)] Q]$ using the metalevel facilities of Maude 2.0. As mentioned in Section 4, these terms, which are of sort `TTrm`, can be rewritten only once. The term of sort `TraceTrm` obtained by rewriting contains a finite trace as a prefix. To create the set of all traces, we compute all possible one-step rewrites. This computation is done at the metalevel by the function `TTrmtoNormalTraceSet` that uses two auxiliary functions `TTrmtoTraceSet` and `TraceSettoNormalTraceSet`.

```
op TTrmtoTraceSet : Term -> TermSet .
op TraceSettoNormalTraceSet : TermSet -> TermSet .
op TTrmtoNormalTraceSet : Term -> TermSet .

eq TTrmtoNormalTraceSet(T) = TraceSettoNormalTraceSet(TTrmtoTraceSet(T)) .
```

The function `TTrmTraceSet` uses the function `allOneStepAux(T,N)` that returns the set of all one-step rewrites (according to the rules in Sections 3 and 4, which are defined in modules named `APISEMATICS` and `APITRACE`, see Figure A.1 in appendix) of the term T which is the metarepresentation of a term of sort `TTrm`, skipping the first N solutions. In the following equations, the operator `_u_` stands for set union.

Notice the use of the operation `metaSearch`, which receives as arguments the metarepresented module to work in, the starting term for search, the pattern to search for, a side condition (empty in this case), the kind of search (which may be `'*` for zero or more rewrites, `'+` for one or more rewrites, and `'!` for only matching normal forms), the depth of search, and the required solution number. It returns the term matching the pattern, its type, and

the substitution produced by the match; to keep only the term, we use the projection `getTerm`.

```

op APITRACE-MOD : -> Module .
eq APITRACE-MOD = ['APITRACE] .
var N : MachineInt .      vars T X : Term .

op allOneStepAux : Term MachineInt Term -> TermSet .
op TraceTermToTrace : Term -> Term .

eq TTrmToTraceSet(T) = allOneStepAux(T,0,'X:TraceTrm) .
eq allOneStepAux(T,N,X) =
    if metaSearch(APITRACE-MOD,T,X,nil,'+,1,N) == failure
    then 'epsilon.Trace
    else TraceTermToTrace(getTerm(metaSearch(APITRACE-MOD,T,X,nil,'+,1,N)))
        u allOneStepAux(T,N + 1,X) fi .

```

The function `TraceTrmToTrace` (whose equations are not shown), used in `allOneStepAux`, extracts the trace $a_1.a_2\dots a_n$ out of a metarepresentation of a term of sort `TraceTrm` of the form $\{a_1\}\{a_2\}\dots\{a_n\}TT$. The function `TraceSettoNormalTraceSet` uses the metalevel operation `metaReduce` to convert each trace in a trace set to its α -normal form. The operation `metaReduce` takes as arguments a metarepresented module and a metarepresented term in that module, and returns the metarepresentation of the fully reduced form of the given term using the equations in the given module, together with its corresponding sort or kind. Again, the projection `getTerm` leaves only the resulting term.

```

eq TraceSettoNormalTraceSet(mt) = mt .
eq TraceSettoNormalTraceSet(T u TS) =
    getTerm(metaReduce(TRACE-MOD,'[ _ ] [ T ]))
    u TraceSettoNormalTraceSet(TS) .

```

We implement the relation \precsim on sets defined in Section 5 as the predicate \ll . We check if $P \sqsubseteq Q$ by computing this predicate on the metarepresented trace sets $\|P\|_{fn(P,Q)}$ and $\|Q\|_{fn(P,Q)}$ as follows. For each (metarepresented) trace T in $\|P\|_{fn(P,Q)}$, we compute the reflexive transitive closure of T with respect to the laws shown in Table 3. The laws are implemented as rewrite rules in the module `TRACE-PREORDER`. We then use the fact that $\|Q\|_{fn(P,Q)} \precsim \|P\|_{fn(P,Q)}$ if and only if for every trace T in $\|P\|_{fn(P,Q)}$ the closure of T and $\|Q\|_{fn(P,Q)}$ have a common element.

```

op TRACE-PREORDER-MOD : -> Module .
eq TRACE-PREORDER-MOD = ['TRACE-PREORDER] .
var N : MachineInt .      vars T T1 T2 X : Term .
var TS TS1 TS2 : TermSet .

op _<<_ : TermSet TermSet -> Bool .
op _<<<_ : TermSet Term -> Bool .
op TTraceClosure : Term -> TermSet .
op TTraceClosureAux : Term Term MachineInt -> TermSet .
op _maypre_ : Term Term -> Bool .

```

```

eq TS2 << mt = true .
eq TS2 << (T1 u TS1) = TS2 <<< T1 and TS2 << TS1 .
eq TS2 <<< T1 = not disjoint?(TS2 , TTraceClosure(T1)) .
eq T1 maypre T2 = TTrmtoNormalTraceSet(T2) << TTrmtoNormalTraceSet(T1) .

```

The computation of the closure of T is done by the function `TTraceClosure`. It uses `TTraceClosureAux` to compute all possible (multi-step) rewrites of the term T using the rules defined in the module `TRACE-PREORDER`, again by means of the metalevel operation `metaSearch`.

```

eq TTraceClosure(T) = TTraceClosureAux(T,'TT:TTrace,0) .
eq TTraceClosureAux(T,X,N) =
  if metaSearch(TRACE-PREORDER-MOD,T,X,nil,'*,maxMachineInt,N) == failure
  then mt
  else getTerm(metaSearch(TRACE-PREORDER-MOD,T,X,nil,'*,maxMachineInt,N))
    u TTraceClosureAux(T,X,N + 1) fi .

```

This computation is terminating as the number of traces to which a trace can rewrite using the trace preorder laws is finite modulo α -equivalence. This follows from the fact that the length of a trace is non-increasing across rewrites, and the free names in the target of a rewrite are also free names in the source. Since the closure of a trace is finite, `metaSearch` can be used to enumerate all the traces in the closure. Note that although the closure of a trace is finite, it is possible to have an infinite rewrite that loops within a subset of the closure. Further, since T is a metarepresentation of a trace, `metaSearch` can be applied directly to T inside the function `TTraceClosureAux(T,X,N)`.

We end this section with a small example, which checks for the may-testing preorder between the processes $P = a(u).b(v).(\nu w)(\bar{w}v|\bar{a}u) + b(u).a(v).(\bar{b}u|\bar{b}w)$ and $Q = b(u).(\bar{b}u|\bar{b}w)$. We define constants `TP` and `TQ` of sort `TTrm`, along with the following equations:

```

eq TP = [[{'a{0}' 'b{0}' 'w{0}'}
           'a{0}('u) . 'b{0}('v) . new['w]('w{0} < 'v{0} > | 'a{0} < 'u{0} >)
           + 'b{0}('u) . 'a{0}('v) . ('b{0} < 'u{0} > | 'b{0} < 'w{0} >)]

```



```

eq TQ = [[{'a{0}' 'b{0}' 'w{0}'}
           'b{0}('u) . ('b{0} < 'u{0} > | 'b{0} < 'w{0} >)]

```

The metarepresentation of these `TTrms` can now be obtained by using `'TP.TTrm` and `'TQ.TTrm`, and we can then check for the may-testing preorder between the given processes as follows:

```

Maude> red 'TP.TTrm maypre 'TQ.TTrm .
reduce in APITRACESET : 'TP.TTrm maypre 'TQ.TTrm .
rewrites: 791690 in 2140ms cpu (2160ms real) (361422 rewrites/second)
result Bool: true
Maude> red 'TQ.TTrm maypre 'TP.TTrm .
reduce in APITRACESET : 'TQ.TTrm maypre 'TP.TTrm .
rewrites: 664833 in 1620ms cpu (1640ms real) (410390 rewrites/second)
result Bool: false

```

Thus, we have $P \sqsubseteq Q$, but $Q \not\sqsubseteq P$. The reader can check that indeed, $\|Q\|_{fn(P,Q)} \lesssim \|P\|_{fn(P,Q)}$, but $\|P\|_{fn(P,Q)} \not\lesssim \|Q\|_{fn(P,Q)}$.

7 Conclusions and Future Work

In this paper, we have described an executable specification in Maude of the operational semantics of an asynchronous version of the π -calculus using conditional rewrite rules with rewrites in the conditions as proposed by Verdejo and Martí-Oliet in [18], and the CINNI calculus proposed by Stehr in [14] for managing names and their binding. In addition, we also implemented the may-testing preorder for π -calculus processes using the Maude metalevel, where we use the `metaSearch` operation to calculate the set of all traces for a process and then compare two sets of traces according to a preorder relation between traces. As emphasized throughout the paper, the new features introduced in Maude 2.0 have been essential for the development of this executable specification, including rewrites in conditions, the `frozen` attribute, and the `metaSearch` operation.

An interesting direction of further work is to extend our implementation to the various typed variants of π -calculus. Two specific typed asynchronous π -calculi for which the work is under way are the local π -calculus ($L\pi$) [10] and the Actor model [1,15]. Both of these formal systems have been used extensively in formal specification and analysis of concurrent object-oriented languages [2,8], and open distributed and mobile systems [9]. The alternate characterization of may testing for both of these typed calculi was recently published [16,17]. We are extending the work presented here to account for the type systems for these calculi, and modifications to the trace based characterization of may testing. We are also looking for interesting concrete applications to which this can be applied; such experiments may require extending our implementation to extensions of π -calculus with higher level constructs, although these may just be syntactic sugar.

Acknowledgements

This research has been supported in part by the Defense Advanced Research Projects Agency (contract numbers F30602-00-2-0586 and F33615-01-C-1907), the ONR MURI Project *A Logical Framework for Adaptive System Interoperability*, and the Spanish CICYT project *Desarrollo Formal de Sistemas Basados en Agentes Móviles* (TIC2000-0701-C02-01). This work was done while the last author was visiting the Department of Computer Science in the University of Illinois at Urbana-Champaign, for whose hospitality he is very grateful. We would like to thank José Meseguer for encouraging us to put together several complementary lines of work in order to get the results described in this paper.

References

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

- [2] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.
- [3] M. Boreale and R. De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120:279–303, 1995.
- [4] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. In *Proceedings 14th IEEE Symposium on Logic in Computer Science, LICS'99, Trento, Italy, July 2–5, 1999*, pages 157–166. IEEE Computer Society Press, 1999.
- [5] M. Boreale, R. De Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes. *Information and Computation*, 172(2):139–164, 2002.
- [6] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 35–50. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>.
- [7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 297–318. Elsevier, 2000. <http://www.elsevier.nl/locate/entcs/volume36.html>.
- [8] I. A. Mason and C. Talcott. A semantically sound actor translation. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, July 7–11, 1997, Proceedings*, volume 1256 of *Lecture Notes in Computer Science*, pages 369–378. Springer-Verlag, 1997.
- [9] M. Merro, J. Kleist, and U. Nestmann. Local π -calculus at work: Mobile objects as mobile processes. In J. van Leeuwen et al., editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000 Sendai, Japan, August 17–19, 2000, Proceedings*, volume 1872 of *Lecture Notes in Computer Science*, pages 390–408. Springer-Verlag, 2000.
- [10] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In K. G. Larsen, S. Skjum, and G. Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13–17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 856–867. Springer-Verlag, 1998.
- [11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.
- [12] R. De Nicola and M. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.

- [13] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, September 1981.
- [14] M.-O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ -, ς - and π -calculi. In K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71–92. Elsevier, 2000. <http://www.elsevier.nl/locate/entcs/volume36.html>.
- [15] C. Talcott. An actor rewriting theory. In J. Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 360–383. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>.
- [16] P. Thati, R. Ziaeи, and G. Agha. A theory of may testing for actors. In B. Jacobs and A. Rensink, editors, *Proceedings IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002), March 20–22, 2002, Enschede, The Netherlands*, pages 147–162. Kluwer Academic Publishers, 2002.
- [17] P. Thati, R. Ziaeи, and G. Agha. A theory of may testing for asynchronous calculi with locality and no name matching. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9–13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
- [18] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In U. Montanari, editor, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. (This volume.) <http://www.elsevier.nl/locate/entcs/volume71.html>.
- [19] P. Viry. Input/output for ELAN. In J. Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 51–64. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>.

A Appendix

The diagram in Figure A.1 illustrates the graph of module importation in our implementation that closely follows the structure of the paper. The complete code is available at <http://osl.cs.uiuc.edu/~ksen/api/>. Here we only show the module that contains the rewrite rules for the operational semantics

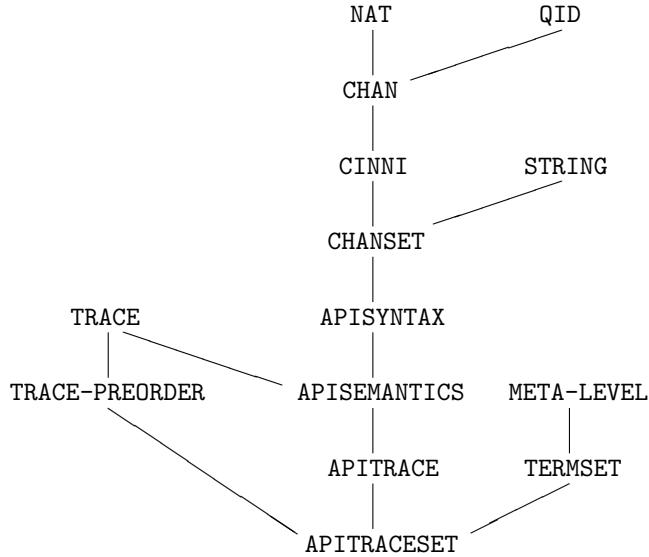


Fig. A.1. The graph of module importation in the implementation.

of asynchronous π -calculus (Table 2). The function `genQid` used in the condition of the last `Res` rule generates an identifier that is fresh, i.e. an identifier not used to construct channel names in the set passed as the argument to the function.

```

mod APISEMANTICS is
  inc APISYNTAX .
  inc CHANSET .
  inc TRACE .
  sorts EnvTrm TraceTrm .
  subsort EnvTrm < TraceTrm .

  op  []_ : Chanset Trm -> EnvTrm [frozen] .
  op  {}_ : Action TraceTrm -> TraceTrm [frozen] .
  op  notinfn : Qid Trm -> Prop .

  vars N : Nat .           vars X Y Z : Qid .
  vars CX CY : Chan .     var CS CS1 CS2 : Chanset .
  vars A : Action .        vars P1 Q1 P Q : Trm .
  var SUM : SumTrm .       var IO : ActionType .

  eq  notinfn(X,P) = not X{0} in freenames(P) .

  rl [Inp] : [CY CS] (CX(X) . P) =>
             {f(i,CX,CY)} ([CY CS] ([X := CY] P)) .

  rl [Inp] : [CY CS] ((CX(X) . P) + SUM) =>
             {f(i,CX,CY)} ([CY CS] ([X := CY] P)) .

  rl [Tau] : [CS] (tau . P) => { tauAct } ([CS] P) .

  rl [Tau] : [CS] ((tau . P) + SUM) => { tauAct } ([CS] P) .
  
```

```

crl [BInp] : [CS] P => {b(i,CX,'u)} ['u{0} [shiftup 'u] CS] P1
    if (not flag in CS) /\
        CS1 := flag 'u{0} [shiftup 'u] CS /\
            [CS1] [shiftup 'u] P => {f(i,CX,'u{0})} [CS1] P1 .

r1 [Out] : [CS] CX < CY > => { f(o,CX,CY) } ([CS] nil) .

crl [Par] : [CS] (P | Q) => {f(IO,CX,CY)} ([CS] (P1 | Q))
    if [CS] P => {f(IO,CX,CY)} ([CS] P1) .

crl [Par] : [CS] (P | Q) =>
    {b(IO,CX,Y)} [Y{0} ([shiftup Y] CS)] (P1 | [shiftup Y] Q)
    if [CS] P => {b(IO,CX,Y)} ([CS1] P1) .

crl [Com] : [CS] (P | Q) => {tauAct} ([CS] (P1 | Q1))
    if [CS] P => {f(o,CX,CY)} ([CS] P1) /\
        [CY CS] Q => {f(i,CX,CY)} ([CY CS] Q1) .

crl [Close] : [CS] (P | Q) => {tauAct} [CS] new [Y] (P1 | Q1)
    if [CS] P => {b(o,CX,Y)} [CS1] P1 /\
        [Y{0} [shiftup Y] CS] [shiftup Y] Q =>
            {f(i,CX,Y{0})} [CS2] Q1 .

crl [Res] : [CS] (new [X] P) =>
    {[shiftdown X] f(IO,CX,CY)} [CS] (new [X] P1)
    if CS1 := [shiftup X] CS /\
        [CS1] P => {f(IO,CX,CY)} [CS1] P1 /\
        (not X{0} in (CX CY)) .

crl [Res] : [CS] (new [X] P) => {tauAct} [CS] (new [X] P1)
    if [CS] P => {tauAct} [CS] P1 .

crl [Res] : [CS] (new [X] P) =>
    {[shiftdown X] b(o,CX,Z)} [Z{0} CS] new[X]([ Y := Z{0} ] P1)
    if Z := genQid(X{0} CS freenames(P)) /\
        {[shiftup X] CS] P => {b(o,CX,Y)} [CS1] P1 /\
        X{0} /= CX .

crl [Open] : [CS] (new[X] P) => {[shiftdown X] b(o,CY,X)} [X{0} CS1] P1
    if CS1 := [shiftup X] CS /\
        [CS1] P => {f(o,CY,X{0})} [CS1] P1 /\
        X{0} /= CY .

crl [If] : [CS1] (if CX = CX then P else Q fi) => {A} [CS2] P1
    if [CS1] P => {A} [CS2] P1 .

crl [Else] : [CS1] (if CX = CY then P else Q fi) => {A} [CS2] Q1
    if CX /= CY /\
        [CS1] Q => {A} [CS2] Q1 .

crl [Rep] : [CS1] (! P) => {A} [CS2] P1
    if [CS1] (P | (! P)) => {A} [CS2] P1 .

endm

```

Generating Optimal Linear Temporal Logic Monitors by Coinduction

Koushik Sen, Grigore Roșu, Gul Agha

Department of Computer Science,

University of Illinois at Urbana-Champaign.

{ksen,grosu,agha}@cs.uiuc.edu

Abstract. A coinduction-based technique to generate an optimal monitor from a Linear Temporal Logic (LTL) formula is presented in this paper. Such a monitor receives a sequence of states (one at a time) from a running process, checks them against a requirements specification expressed as an LTL formula, and determines whether the formula has been violated or validated. It can also say whether the LTL formula is not monitorable any longer, i.e., that the formula can in the future neither be violated nor be validated. A Web interface for the presented algorithm adapted to extended regular expressions is available.

1 Introduction

Linear Temporal Logic (LTL) [19] is a widely used logic for specifying properties of reactive and concurrent systems. The models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests, operating system being typical example. LTL has been mainly used to specify properties of finite-state reactive and concurrent systems, so that the full correctness of the system can be verified automatically, using model checking or theorem proving. Model checking of programs has received an increased attention from the formal methods community within the last couple of years, and several tools have emerged that directly model check source code written in Java or C [7, 26, 27]. Unfortunately, such formal verification techniques are not scalable to real-sized systems without exerting a substantial effort to abstract the system more or less manually to a model that can be analyzed.

Testing scales well, and in practice it is by far the technique most used to validate software systems. Our approach follows research which merges testing and temporal logic specification in order to achieve some of the benefits of both approaches; we avoid some of the pitfalls of ad hoc testing as well as the complexity of full-blown theorem proving and model checking. While this merger provides a scalable technique, it does result in a loss of coverage: the technique may be used to examine a single execution trace at a time, and may not be used to *prove* a system correct. Our work is based on the observation that software engineers are willing to trade coverage for scalability, so our goals is relatively conservative: we provide tools that use formal methods in a lightweight manner, use traditional programming languages or underlying executional engines (such as JVMs), are completely automatic, implement very efficient algorithms, and can help find *many* errors in programs.

Recent trends suggest that the software analysis community is interested in scalable techniques for software verification. Earlier work by Havelund and

Roşu [10] proposed a method based on merging temporal logics and testing. The Temporal Rover tool (TR) and its successor DB Rover by Drusinsky [2] have been commercialized. These tools instrument the Java code so that it can check the satisfaction of temporal logic properties at runtime. The MaC tool by Lee et al. [14, 17] has been developed to monitor safety properties in interval past time temporal logic. In works by O’Malley et al. and Richardson et al. [20, 21], various algorithms to generate testing automata from temporal logic formulae, are described. Java PathExplorer [8] is a runtime verification environment currently under development at NASA Ames. It can analyze a single execution trace. The Java MultiPathExplorer tool [25] proposes a technique to monitor all equivalent traces that can be extracted from a given execution, thus increasing the coverage of monitoring. Giannakopoulou et al. and Havelund et al. in [4, 9] propose efficient algorithms for monitoring future time temporal logic formulae, while Havelund et al. in [11] gives a technique to synthesize efficient monitors from past time temporal formulae. Roşu et al. in [23] shows use of rewriting to perform runtime monitoring of extended regular expressions. An approach similar to this paper is used to generate optimal monitors for extended regular expressions in work by Sen et al. [24].

In this paper, we present a new technique based on the modern coalgebraic method to generate optimal monitors for LTL formulae. In fact, such monitors are the minimal deterministic finite automata required to do the monitoring. Our current work makes two major contributions. First, we give a coalgebraic formalization of LTL and show that coinduction is a viable and reasonably practical method to prove monitoring-equivalences of LTL formulae. Second, building on the coinductive technique, we present an algorithm to directly generate minimal deterministic automata from an LTL formula. Such an automaton may be used to monitor good or bad prefixes of an execution trace (this notion will be rigorously formalized in subsequent sections).

We describe the monitoring as synchronous and deterministic to obtain *minimal* good or bad prefixes. However, if the cost of such monitoring is deemed too high in some application, and one is willing to tolerate some delay in discovering violations, the same technique could be applied on the traces intermittently – in which case one would not get minimal good or bad prefixes but could either bound the delay in discovering violations, or guarantee eventual discovery. We also give lower and upper bounds on the size of such automata.

The closely related work by Geilen [3] builds monitors to detect a subclass of bad and good prefixes, which are called *informative bad and good prefixes*. Using a tableau-based technique, [3] can generate monitors of exponential size for informative prefixes. In our approach, we generate minimal monitors for detecting all kinds of bad and good prefixes. This generality comes at a price: the size of our monitors can be doubly exponential in the worst case, and this complexity cannot be avoided.

One standard way to generate an optimal monitor is to use the Büchi automata construction [16] for LTL to generate a non-deterministic finite automaton, determinize it and then to minimize it. In this method, one checks only

the syntactic equivalence of LTL formulae. In the coalgebraic technique that we propose as an alternative method, we make use of the *monitoring equivalence* (defined in subsequent sections) of LTL formulae. We thus obtain the minimal automaton *in a single go* and minimize the usage of computational space. Moreover, our technique is completely based on deductive methods and can be applied to any logic or algebra for which there is a suitable behavioral specification. A related application can be found in [24] in which the minimal deterministic finite automata for extended regular expressions is generated.

2 Linear Temporal Logic and Derivatives

In order to make the paper self-contained, we briefly describe classical Linear Temporal Logic over infinite traces. We use the classical definition of Linear Temporal Logic and assume a finite set AP of atomic propositions. The syntax of LTL is as follows:

$$\begin{aligned} \phi ::= & \text{ true } | \text{ false } | a \in AP | \neg\phi | \phi \wedge \phi | \phi \vee \phi | \phi \rightarrow \phi | \phi \equiv \phi | \phi \oplus \phi & \text{ propositional} \\ & \phi \mathcal{U} \phi | \circ\phi | \square\phi | \diamond\phi & \text{ temporal} \end{aligned}$$

The semantics of LTL is given for infinite traces. An infinite trace is an infinite sequence of program states, each state denoting the set of atomic propositions that hold at that state. The atomic propositions that hold in a given state s is given by $AP(s)$. We denote an infinite trace by ρ ; $\rho(i)$ denotes the i -th state in the trace and ρ^i denotes the suffix of the trace ρ starting from the i -th state. The notion that an infinite trace ρ *satisfies* a formula ϕ is denoted by $\rho \models \phi$, and is defined inductively as follows:

$$\begin{array}{ll} \rho \models \text{true for all } \rho & \rho \not\models \text{false for all } \rho \\ \rho \models a \text{ iff } a \in AP(\rho(1)) & \rho \models \neg\phi \text{ iff } \rho \not\models \phi \\ \rho \models \phi_1 \vee \phi_2 \text{ iff } \rho \models \phi_1 \text{ or } \rho \models \phi_2 & \rho \models \phi_1 \wedge \phi_2 \text{ iff } \rho \models \phi_1 \text{ and } \rho \models \phi_2 \\ \rho \models \phi_1 \oplus \phi_2 \text{ iff } \rho \models \phi_1 \text{ exclusive or } \rho \models \phi_2 & \rho \models \phi_1 \rightarrow \phi_2 \text{ iff } \rho \models \phi_1 \text{ implies } \rho \models \phi_2 \\ \rho \models \phi_1 \equiv \phi_2 \text{ iff } \rho \models \phi_1 \text{ iff } \rho \models \phi_2 & \rho \models \circ\phi \text{ iff } \rho^2 \models \phi \\ \rho \models \square\phi \text{ iff } \forall j \geq 1 \rho^j \models \phi & \rho \models \diamond\phi \text{ iff } \exists j \geq 1 \text{ such that } \rho^j \models \phi \\ \rho \models \phi_1 \mathcal{U} \phi_2 \text{ iff there exists a } j \geq 1 \text{ such that } & \rho^j \models \phi_2 \text{ and } \forall 1 \leq i < j : \rho^i \models \phi_1 \end{array}$$

The set of all infinite traces that satisfy the formula ϕ is called the language expressed by the formula ϕ and is denoted by L_ϕ . Thus, $\rho \in L_\phi$ if and only if $\rho \models \phi$. The language L_ϕ is also called the *property* expressed by the formula ϕ . We informally say that an infinite trace ρ satisfies a property ϕ iff $\rho \models \phi$.

A property in LTL can be seen as the intersection of a *safety* property and a *liveness* property [1]. A property is a liveness property if for every finite trace α there exists an infinite trace ρ such that $\alpha.\rho$ satisfies the property. A property is a safety property if for every infinite trace ρ not satisfying the property, there exists a finite prefix α such that for all infinite traces ρ' , $\alpha.\rho'$ does not satisfy the property. The prefix α is called a *bad prefix* [3]. Thus, we say that a finite prefix α is a bad prefix for a property if for all infinite traces ρ , $\alpha.\rho$ does not satisfy the property. On the other hand, a *good prefix* for a property is a prefix α such that for all infinite traces ρ , $\alpha.\rho$ satisfies the property. A bad or a good prefix can also be *minimal*. We say that a bad (or a good prefix) α is minimal if α is a bad (or good) prefix and no finite prefix α' of α is bad (or good) prefix.

We use a novel coinduction-based technique to generate an optimal monitor that can detect good and bad prefixes incrementally for a given trace. The essential idea is to process, one by one, the states of a trace as these states are generated; at each step the process checks if the finite trace that we have already generated is a minimal good prefix or a minimal bad prefix. At any point, if we find that the finite trace is a minimal bad prefix, we say that the property is violated. If the finite trace is a minimal good prefix then we stop monitoring for that particular trace and say that the property holds for that trace.

At any step, we will also detect if it is not possible to monitor a formula any longer. We may stop monitoring at that point and say the trace is no longer *monitorable* and save the monitoring overhead. Otherwise, we continue by processing one more state and appending that state to the finite trace. We will see in the subsequent sections that these monitors can report a message as soon as a good or a bad prefix is encountered; therefore, the monitors are synchronous. Two more variants of the optimal monitor are also proposed; these variants can be used to efficiently monitor either bad prefixes or good prefixes (rather than both). Except in degenerate cases, such monitors have smaller sizes than the monitors that can detect both bad and good prefixes.

In order to generate the minimal monitor for an LTL formula, we will use several notions of equivalence for LTL:

Definition 1 (\equiv). *We say that two LTL formulae ϕ_1 and ϕ_2 are equivalent i.e. $\phi_1 \equiv \phi_2$ if and only if $L_{\phi_1} = L_{\phi_2}$.*

Definition 2 (\equiv_B). *For a finite trace α we say that $\alpha \not\models \phi$ iff α is bad prefix for ϕ i.e. for every infinite trace ρ it is the case that $\alpha.\rho \notin L_\phi$. Given two LTL formulae ϕ_1 and ϕ_2 , ϕ_1 and ϕ_2 are said to be bad prefix equivalent i.e. $\phi_1 \equiv_B \phi_2$ if and only if for every finite trace α , $\alpha \not\models \phi_1$ iff $\alpha \not\models \phi_2$.*

Definition 3 (\equiv_G). *For a finite trace α we say that $\alpha \models \phi$ iff α is good prefix for ϕ i.e. for every infinite trace ρ it is the case that $\alpha.\rho \in L_\phi$. Given two LTL formulae ϕ_1 and ϕ_2 , ϕ_1 and ϕ_2 are said to be good prefix equivalent i.e. $\phi_1 \equiv_G \phi_2$ if and only if for every finite trace α , $\alpha \models \phi_1$ iff $\alpha \models \phi_2$.*

Definition 4 (\equiv_{GB}). *We say that ϕ_1 and ϕ_2 are good-bad prefix equivalent i.e. $\phi_1 \equiv_{GB} \phi_2$ if and only if $\phi_1 \equiv_B \phi_2$ and $\phi_1 \equiv_G \phi_2$.*

Thus, for our purpose, the two non equivalent formulae $\Box\Diamond\phi$ and $\Diamond\Box\phi$ are good-bad prefix equivalent since they do not have any good or bad prefixes. Such formula are not monitorable. Note that the equivalence relation \equiv is included in the equivalence relation \equiv_{GB} , which is in turn included in both \equiv_G and \equiv_B . We will use the equivalences \equiv_G, \equiv_B , and \equiv_{GB} to generate optimal monitors that detect good prefixes only, bad prefixes only and both bad and good prefixes respectively. We call these three equivalences *monitoring equivalences*.

2.1 Derivatives

We describe the notion of derivatives for LTL [9, 10] based on the idea of *state consumption*: an LTL formula ϕ and a state s generate another LTL formula, denoted by $\phi\{s\}$, with the property that for any finite trace α , $s\alpha \not\models \phi$ if and only if $\alpha \not\models \phi\{s\}$ and $s\alpha \models \phi$ if and only if $\alpha \models \phi\{s\}$. We define the operator $\{-\}$ recursively through the following equations:

$$\begin{array}{ll}
\text{false } \{s\} = \text{ false} & \text{true } \{s\} = \text{ true} \\
p\{s\} = \text{ if } p \in AP(s) \text{ then true else false} & (\neg\phi)\{s\} = \neg(\phi\{s\}) \\
(\phi_1 \vee \phi_2)\{s\} = \phi_1\{s\} \vee \phi_2\{s\} & (\phi_1 \wedge \phi_2)\{s\} = \phi_1\{s\} \wedge \phi_2\{s\} \\
(\phi_1 \rightarrow \phi_2)\{s\} = \phi_1\{s\} \rightarrow \phi_2\{s\} & (\phi_1 \oplus \phi_2)\{s\} = \phi_1\{s\} \oplus \phi_2\{s\} \\
(\Diamond\phi)\{s\} = \phi\{s\} \vee \Diamond\phi & (\Box\phi)\{s\} = \phi\{s\} \wedge \Box\phi \\
(\phi_1 \mathcal{U} \phi_2)\{s\} = \phi_2\{s\} \vee (\phi_1\{s\} \wedge \phi_1 \mathcal{U} \phi_2) &
\end{array}$$

We use the decision procedure for propositional calculus by Hsiang [13] to get a canonical form for a propositional formula. The procedure reduces a tautological formula to the constant true, a false formula to the constant false, and all other formulae to canonical forms modulo associativity and commutativity. An unusual aspect of this procedure is that the canonical forms consist of exclusive or (\oplus) of conjunctions. The procedure is given below using equations that are shown to be Church-Rosser and terminating modulo associativity and commutativity.

$$\begin{array}{ll}
\text{true } \wedge \phi = \phi & \text{false } \wedge \phi = \text{ false} \\
\phi \wedge \phi = \phi & \text{false } \oplus \phi = \phi \\
\phi \oplus \phi = \text{ false} & \neg\phi = \text{ true } \oplus \phi \\
\phi_1 \wedge (\phi_2 \oplus \phi_3) = (\phi_1 \wedge \phi_2) \oplus (\phi_1 \wedge \phi_3) & \phi_1 \vee \phi_2 = (\phi_1 \wedge \phi_2) \oplus \phi_1 \oplus \phi_2 \\
\phi_1 \rightarrow \phi_2 = \text{ true } \oplus \phi_1 \oplus (\phi_1 \wedge \phi_2) & \phi_1 \equiv \phi_2 = \text{ true } \oplus \phi_1 \oplus \phi_2
\end{array}$$

The exclusive or operator \oplus and the \wedge operator are defined as commutative and associative. The equations DERIVATIVE and PROPOSITIONAL CALCULUS when regarded as rewriting rules are terminating and Church-Rosser (modulo associativity and commutativity of \wedge and \oplus), so they can be used as a functional procedure to calculate derivatives.

In the rest of the paper, at several places we need to check if an LTL formula is equivalent to true or false. This can be done using the tableau-based proof method for LTL; the STeP tool at Stanford [18] has such an implementation.

The following result gives a way to determine if a prefix is good or bad for a formula through derivations.

Theorem 1. *a) For any LTL formula ϕ and for any finite trace $\alpha = s_1 s_2 \dots s_n$, α is a bad prefix for ϕ if and only if $\phi\{s_1\}\{s_2\}\dots\{s_n\} \equiv \text{false}$. Similarly, α is a good prefix for ϕ if and only if $\phi\{s_1\}\{s_2\}\dots\{s_n\} \equiv \text{true}$. b) The formula $\phi\{s_1\}\{s_2\}\dots\{s_n\}$ needs $O(2^{\text{size}(\phi)})$ space to be stored.*

Proof. b): Due to the Boolean ring equations above regarded as simplification rules, any LTL formula is kept in a canonical form, which is an exclusive disjunction of conjunctions, where conjuncts have temporal operators at top. Moreover, after a series of applications of derivatives s_1, s_2, \dots, s_n , the conjuncts in the normal form $\phi\{s_1\}\{s_2\}\dots\{s_n\}$ are subterms of the initial formula ϕ , each having a temporal operator at its top. Since there are at most $\text{size}(\phi)$ such subformulae, it follows that there are at most $2^{\text{size}(\phi)}$ possibilities to combine them in a conjunction. Therefore, one needs space $O(2^{\text{size}(\phi)})$ to store any exclusive disjunction of such conjunctions. This reasoning only applies on “idealistic” rewriting engines, which carefully optimize space needs during rewriting. \square

In order to effectively generate optimal monitors, it is crucial to detect efficiently and as early as possible when two derivatives are equivalent. In the rest of the paper we use coinductive techniques to solve this problem. We define the

operators $G : LTL \rightarrow \{\text{true}, \text{false}\}$ and $B : LTL \rightarrow \{\text{true}, \text{false}\}$ that return true if an LTL formula is equivalent (\equiv) to true or false respectively, and return false otherwise. We define an operator $GB : LTL \rightarrow \{0, 1, ?\}$ that checks if an LTL formula ϕ is equivalent to false or true and returns 0 or 1, respectively, and returns ? if the formula is not equivalent to either true or false.

3 Hidden Logic and Coinduction

We use circular coinduction, defined rigorously in the context of hidden logics and implemented in the BOBJ system [22, 5, 6], to test whether two LTL formulae are good-bad prefix equivalent. A particularly appealing aspect of circular coinduction in the framework of LTL formula is that it not only shows that two LTL formulae are good-bad prefix equivalent, but also generates a larger set of good-bad prefix equivalent LTL formulae which will all be used in order to generate the target monitor. Readers familiar with circular coinduction may assume the result in Theorem 4 and read Section 4 concurrently.

Hidden logic is a natural extension of algebraic specification which benefits of a series of generalizations in order to capture various natural notions of behavioral equivalence found in the literature. It distinguishes *visible* sorts for data from *hidden* sorts for states, with states *behaviorally equivalent* if and only if they are indistinguishable under a formally given set of experiments. In order to keep the presentation simple and self-contained, we define a simplified version of hidden logic together with its associated circular coinduction proof rule which is nevertheless general enough to support the definition of LTL formulae and prove that they are behaviorally good and/or bad prefix equivalent.

3.1 Algebraic Preliminaries

We assume that the reader is familiar with basic equational logic and algebra but recall a few notions in order to just make our notational conventions precise. An S -sorted signature Σ is a set of sorts/types S together with operational symbols on those, and a Σ -algebra A is a collection of sets $\{A_s \mid s \in S\}$ and a collection of functions appropriately defined on those sets, one for each operational symbol. Given an S -sorted signature Σ and an S -indexed set of variables Z , let $T_\Sigma(Z)$ denote the Σ -term algebra over variables in Z . If $V \subseteq S$ then $\Sigma|_V$ is a V -sorted signature consisting of all those operations in Σ with sorts entirely in V . We may let $\sigma(X)$ denote the term $\sigma(x_1, \dots, x_n)$ when the number of arguments of σ and their order and sorts are not important. If only one argument is important, then to simplify writing we place it at the beginning; for example, $\sigma(t, X)$ is a term having σ as root with no important variables as arguments except one, in this case t . If t is a Σ -term of sort s' over a special variable $*$ of sort s and A is a Σ -algebra, then $A_t : A_s \rightarrow A_{s'}$ is the usual interpretation of t in A .

3.2 Behavioral Equivalence, Satisfaction and Specification

Given disjoint sets V, H called *visible* and *hidden sorts*, a *hidden (V, H) -signature*, say Σ , is a many sorted $(V \cup H)$ -signature. A *hidden subsignature* of Σ is a hidden (V, H) -signature Γ with $\Gamma \subseteq \Sigma$ and $\Gamma|_V = \Sigma|_V$. The *data signature* is $\Sigma|_V$. An

operation of visible result not in $\Sigma \upharpoonright V$ is called an *attribute*, and a hidden sorted operation is called a *method*.

Unless otherwise stated, the rest of this section assumes fixed a hidden signature Σ with a fixed subsignature Γ . Informally, Σ -algebras are universes of possible states of a system, i.e., “black boxes,” for which one is only concerned with behavior under experiments with operations in Γ , where an experiment is an observation of a system attribute after perturbation.

A Γ -context for sort $s \in V \cup H$ is a term in $T_\Gamma(\{*\} : s)$ with one occurrence of $*$. A Γ -context of visible result sort is called a Γ -experiment. If c is a context for sort h and $t \in T_{\Sigma,h}$ then $c[t]$ denotes the term obtained from c by substituting t for $*$; we may also write $c[*]$ for the context itself.

Given a hidden Σ -algebra A with a hidden subsignature Γ , for sorts $s \in (V \cup H)$, we define Γ -behavioral equivalence of $a, a' \in A_s$ by $a \equiv_\Sigma^\Gamma a'$ iff $A_c(a) = A_c(a')$ for all Γ -experiments c ; we may write \equiv instead of \equiv_Σ^Γ when Σ and Γ can be inferred from context. We require that all operations in Σ are compatible with \equiv_Σ^Γ . Note that behavioral equivalence is the identity on visible sorts, since the trivial contexts $* : v$ are experiments for all $v \in V$. A major result in hidden logics, underlying the foundations of coinduction, is that Γ -behavioral equivalence is the largest equivalence which is identity on visible sorts and which is compatible with the operations in Γ .

Behavioral satisfaction of equations can now be naturally defined in terms of behavioral equivalence. A hidden Σ -algebra A Γ -behaviorally satisfies a Σ -equation $(\forall X) t = t'$, say e , iff for each $\theta : X \rightarrow A$, $\theta(t) \equiv_\Sigma^\Gamma \theta(t')$; in this case we write $A \models_\Sigma^\Gamma e$. If E is a set of Σ -equations we then write $A \models_\Sigma^\Gamma E$ when A Γ -behaviorally satisfies each Σ -equation in E . We may omit Σ and/or Γ from \models_Σ^Γ when they are clear.

A behavioral Σ -specification is a triple (Σ, Γ, E) where Σ is a hidden signature, Γ is a hidden subsignature of Σ , and E is a set of Σ -sentences equations. Non-data Γ -operations (i.e., in $\Gamma - \Sigma \upharpoonright V$) are called *behavioral*. A Σ -algebra A behaviorally satisfies a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ iff $A \models_\Sigma^\Gamma E$, in which case we write $A \models \mathcal{B}$; also $\mathcal{B} \models e$ iff $A \models \mathcal{B}$ implies $A \models_\Sigma^\Gamma e$.

LTL can be very naturally defined as a behavioral specification. The enormous benefit of doing so is that the behavioral inference, including most importantly coinduction, provide a *decision procedure* for good-bad prefix equivalence.

Example 1. A behavioral specification of LTL defines a set of two visible sorts $V = \{\text{Triple}, \text{State}\}$, one hidden sort $H = \{\text{Ltl}\}$, one behavioral attribute $GB : \text{Ltl} \rightarrow \text{Triple}$ (defined as an operator in Subsection 2.1) and one behavioral method, the derivative, $_{} : \text{Ltl} \times \text{State} \rightarrow \text{Ltl}$, together with all the other operations in Section 2 defining LTL, including the states in S which are defined as visible constants of sort State , and all the equations in Subsection 2.1. The sort *Triple* consists of three constants 0, 1, and ?. We call this the *LTL behavioral specification* and we use $\mathcal{B}_{LTL/GB}$ to denote it.

Since the only behavioral operators are the test for equivalence to true and false and the derivative, it follows that the experiments have exactly the form $GB(*\{s_1\}\{s_2\}\dots\{s_n\})$, for any states s_1, s_2, \dots, s_n . In other words, an experi-

ment consists of a series of derivations followed by an application of the operator GB , and therefore two LTL formulae are *behavioral equivalent* if and only if they cannot be distinguished by such experiments. Such behavioral equivalence is exactly same as good-bad prefix equivalence. In the specification of $\mathcal{B}_{LTL/GB}$ if we replace the attribute GB by B (or G), as defined in Subsection 2.1, the behavioral equivalence becomes same as bad prefix (or good prefix) equivalence. We denote such specifications by $\mathcal{B}_{LTL/B}$ (or $\mathcal{B}_{LTL/G}$). Notice that the above reasoning applies within *any algebra* satisfying the presented behavioral specification. The one we are interested in is, of course, the *free one*, whose set carriers contain exactly the LTL formulae as presented in Section 2, and the operations have the obvious interpretations. We informally call it the *LTL algebra*.

Letting \equiv_b denote the behavioral equivalence relation generated on the LTL algebra, then Theorem 1 immediately yields the following important result.

Theorem 2. *If ϕ_1 and ϕ_2 are two LTL formulae then $\phi_1 \equiv_b \phi_2$ in $\mathcal{B}_{LTL/GB}$ iff ϕ_1 and ϕ_2 are good-bad prefix equivalent. Similarly, $\phi_1 \equiv_b \phi_2$ in $\mathcal{B}_{LTL/B}$ (or $\mathcal{B}_{LTL/G}$) if and only if ϕ_1 and ϕ_2 are bad prefix (or good prefix) equivalent.*

This theorem allows us to prove good-bad prefix equivalence (or bad prefix or good prefix equivalence) of LTL formulae by making use of behavioral inference in the LTL behavioral specification $\mathcal{B}_{LTL/GB}$ (or $\mathcal{B}_{LTL/B}$ or $\mathcal{B}_{LTL/G}$) including (especially) circular coinduction. The next section shows how circular coinduction works and how it can be used to show LTL formulae good-bad prefix equivalent (or bad prefix equivalent or good prefix equivalent). From now onwards we will refer $\mathcal{B}_{LTL/GB}$ simply by \mathcal{B} .

3.3 Circular Coinduction as an Inference Rule

In the simplified version of hidden logics defined above, the usual equational inference rules, i.e., reflexivity, symmetry, transitivity, substitution and congruence [22] are all sound for behavioral satisfaction. However, equational reasoning can derive only a very limited amount of interesting behavioral equalities. For that reason, *circular coinduction* has been developed as a very powerful automated technique to show behavioral equivalence. We let \Vdash denote the relation being defined by the equational rules plus circular coinduction, for deduction from a specification to an equation.

Before formally defining circular coinduction, we give the reader some intuitions by duality to structural induction. The reader who is only interested in using the presented procedure or who is not familiar with structural induction, can skip this paragraph. Inductive proofs show equality of terms $t(x), t'(x)$ over a given variable x (seen as a constant) by showing $t(\sigma(x))$ equals $t'(\sigma(x))$ for all σ in a basis, while circular coinduction shows terms t, t' behaviorally equivalent by showing equivalence of $\delta(t)$ and $\delta(t')$ for all behavioral operations δ . Coinduction applies behavioral operations at the top, while structural induction applies generator/constructor operations at the bottom. Both induction and circular coinduction assume some “frozen” instances of t, t' equal when checking the inductive/coinductive step: for induction, the terms are frozen at the bottom by replacing the induction variable by a constant, so that no other terms can be placed beneath the induction variable, while for coinduction, the terms are

frozen at the top, so that they cannot be used as subterms of other terms (with some important but subtle exceptions which are not needed here; see [6]).

Freezing terms at the top is elegantly handled by a simple trick. Suppose every specification has a special visible sort b , and for each (hidden or visible) sort s in the specification, a special operation $[.] : s \rightarrow b$. No equations are assumed for these operations and no user defined sentence can refer to them; they are there for technical reasons. Thus, with just the equational inference rules, for any behavioral specification \mathcal{B} and any equation $(\forall X) t = t'$, it is necessarily the case that $\mathcal{B} \Vdash (\forall X) t = t'$ iff $\mathcal{B} \Vdash (\forall X) [t] = [t']$. The rule below preserves this property. Let the sort of t, t' be hidden; then

Circular Coinduction:

$$\frac{\mathcal{B} \cup \{(\forall X) [t] = [t']\} \Vdash (\forall X, W) [\delta(t, W)] = [\delta(t', W)], \text{ for all appropriate } \delta \in \Gamma}{\mathcal{B} \Vdash (\forall X) t = t'}$$

We call the equation $(\forall X) [t] = [t']$ added to \mathcal{B} a **circularity**; it could just as well have been called a coinduction hypothesis or a co-hypothesis, but we find the first name more intuitive because from a coalgebraic point of view, coinduction is all about finding circularities.

Theorem 3. *The usual equational inference rules together with Circular Coinduction are sound. That means that if $\mathcal{B} \Vdash (\forall X) t = t'$ and $\text{sort}(t, t') \neq b$, or if $\mathcal{B} \Vdash (\forall X) [t] = [t']$, then $\mathcal{B} \models (\forall X) t = t'$.*

Circular coinductive rewriting[5, 6] iteratively rewrites proof tasks to their normal forms followed by an one step coinduction if needed. Since the rules in $\mathcal{B}_{LTL/GB}$, $\mathcal{B}_{LTL/B}$, and $\mathcal{B}_{LTL/G}$ are ground Church-Rosser and terminating, this provides us with a decision procedure for good-bad prefix equivalence, bad prefix equivalence, and good prefix equivalence of LTL formulae respectively.

Theorem 4. *If ϕ_1 and ϕ_2 are two LTL formulae, then $\phi_1 \equiv_{GB} \phi_2$ if and only if $\mathcal{B}_{LTL/GB} \Vdash \phi_1 = \phi_2$. Similarly, if ϕ_1 and ϕ_2 are two LTL formulae, then $\phi_1 \equiv_B \phi_2$ (or $\phi_1 \equiv_G \phi_2$) if and only if $\mathcal{B}_{LTL/B} \Vdash \phi_1 = \phi_2$ (or $\mathcal{B}_{LTL/G} \Vdash \phi_1 = \phi_2$). Moreover, circular coinductive rewriting provides us with a decision procedure for good-bad prefix equivalence, bad prefix equivalence, and good prefix equivalence of LTL formulae.*

Proof. By soundness of behavioral reasoning (Theorem 3), one implication follows immediately via Theorem 2. For the other implication, assume that ϕ_1 and ϕ_2 are good-bad prefix equivalent (or good prefix or bad prefix equivalent, respectively) and that the equality $\phi_1 = \phi_2$ is not derivable from $\mathcal{B}_{LTL/GB}$ (or $\mathcal{B}_{LTL/G}$ or $\mathcal{B}_{LTL/B}$, respectively). By Theorem 1, the number of formulae into which any LTL formula can be derived via a sequence of events is finite, which means that the total number of equalities $\phi'_1 = \phi'_2$ that can be derived via the circular coinduction rule is also finite. That implies that the only reason for which the equality $\phi_1 = \phi_2$ *cannot* be proved by circular coinduction is because it is in fact disproved by some experiment, which implies the existence of some events a_1, \dots, a_n such that $GB(\phi_1\{a_1\} \cdots \{a_n\}) \neq GB(\phi_2\{a_1\} \cdots \{a_n\})$ (or the equivalent ones for B or G). However, this is obviously a contradiction because if ϕ_1 and ϕ_2 are good-bad (or good or bad) prefix equivalent that so are $\phi_1\{a_1\} \cdots \{a_n\}$ and $\phi_2\{a_1\} \cdots \{a_n\}$, and GB (or G or B) preserve this equivalence.

4 Generating Optimal Monitors by Coinduction

We now show how one can use the set of circularities generated by applying the circular coinduction rules in order to generate, from any LTL formula, an optimal monitor that can detect both good and bad prefixes. The optimal monitor thus generated will be a minimal deterministic finite automaton containing two final states true and false. We call such a monitor *GB-automaton*. We conclude the section by modifying the algorithm to generate smaller monitors that can detect either bad or good prefixes. We call such monitors *B-automaton* and *G-automaton* respectively. The main idea behind the algorithm is to associate states in GB-automaton to LTL formulae obtained by deriving the initial LTL formula; when a new LTL formula is generated, it is tested for good-bad prefix equivalence with all the other already generated LTL formulae by using the coinductive procedure presented in the previous section. A crucial observation which significantly reduces the complexity of our procedure is that once a good-bad prefix equivalence is proved by circular coinductive rewriting, the entire set of circularities accumulated represent good-bad prefix equivalent LTL formulae. These can be used to quickly infer the other good-bad prefix equivalences, without having to generate the same circularities over and over again.

Since BOBJ does not (yet) provide any mechanism to return the set of circularities accumulated after proving a given behavioral equivalence, we were unable to use BOBJ to implement our optimal monitor generator. Instead, we have implemented our own version of coinductive rewriting engine for LTL formulae, which is described below.

We are given an initial LTL formula ϕ_0 over atomic propositions P . Then $\sigma = 2^P$ is the set of possible states that can appear in an execution trace; note that σ will be the set of alphabets in the GB-automaton. Now, from ϕ_0 we want to generate a GB-automaton $D = (S, \sigma, \delta, s_0, \{\text{true}, \text{false}\})$, where S is the set of states of the GB-automaton, $\delta : S \times \sigma \rightarrow S$ is the transition function, s_0 is the initial state of the GB-automaton, and $\{\text{true}, \text{false}\} \subseteq S$ is the set of final states of the DFA. The coinductive rewriting engine explicitly accumulates the proven circularities in a set. The set is initialized to an empty set at the beginning of the algorithm. It is updated with the accumulated circularities whenever we prove good-bad prefix equivalence of two LTL formulae in the algorithm. The algorithm maintains the set of states S in the form of non good-bad prefix equivalent LTL formulae. At the beginning of the algorithm S is initialized with two elements, the constant formulae true and false. Then, we check if the initial LTL formula ϕ_0 is equivalent to true or false. If ϕ_0 is equivalent to true or false, we set s_0 to true or false respectively and return D as the GB-automaton. Otherwise, we set s_0 to ϕ_0 , add ϕ_0 to the set S , and invoke the procedure **dfs** (see Fig 1) on ϕ_0 .

The procedure **dfs** generates the derivatives of a given formula ϕ for all $x \in \sigma$ one by one. A derivative $\phi_x = \phi\{x\}$ is added to the set S , if the set does not contain any LTL formula good-bad prefix equivalent to the derivative ϕ_x . We then extend the transition function by setting $\delta(\phi, x) = \phi_x$ and recursively invoke **dfs** on ϕ_x . On the other hand, if an LTL formula ϕ' equivalent to the derivative already exists in the set S , we extend the transition function by setting

$\delta(\phi, x) = \phi'$. To check if an LTL formula, good-bad prefix equivalent to the derivative ϕ_x , already exists in the set S , we sequentially go through all the elements of the set S and try to prove its good-bad prefix equivalence with ϕ_x . In testing the equivalence we first add the set of circularities to the initial $\mathcal{B}_{LTL/GB}$. Then we invoke the coinductive procedure. If for some LTL formula $\phi' \in S$, we are able to prove that $\phi' \equiv_{GB} \phi_x$ i.e $\mathcal{B}_{LTL/GB} \cup Eq_{all} \cup Eq_{new} \Vdash \phi' = \phi_x$, then we add the new equivalences Eq_{new} , created by the coinductive procedure, to the set of circularities. Thus we reuse the already proven good-bad prefix equivalences in future proofs.

```

 $S \leftarrow \{\text{true}, \text{false}\}$ 
dfs( $\phi$ )
begin
  foreach  $x \in \sigma$  do
     $\phi_x \leftarrow \phi\{x\};$ 
    if  $\exists \phi' \in S$  such that  $\mathcal{B}_{LTL/GB} \cup Eq_{all} \cup Eq_{new} \Vdash \phi' = \phi_x$  then
       $\delta(\phi, x) = \phi'; Eq_{all} \leftarrow Eq_{all} \cup Eq_{new}$ 
    else  $S \leftarrow S \cup \{\phi_x\}; \delta(\phi, x) = \phi_x; \text{dfs}(\phi_x); \text{fi}$ 
  endfor
end

```

Fig. 1. LTL to optimal monitor generation algorithm

The GB-automaton generated by the procedure **dfs** may now contain some states which are non-final and from which the GB-automaton can never reach a final state. We remove these redundant states by doing a breadth first search in backward direction from the final states. This can be done in time linear in the size of the GB-automaton. If the resultant GB-automaton contains the initial state s_0 then we say that the LTL formula is monitorable. That is for the LTL formula to be monitorable there must be path from the initial state to a final state i.e. to true or false state. Note that the GB-automaton may now contain non-final states from which there may be no transition for some $x \in \sigma$. Also note that no transitions are possible from the final states.

The correctness of the algorithm is given by the following theorem.

Theorem 5. *If D is the GB-automaton generated for a given LTL formula ϕ by the above algorithm then*

- 1) $\mathcal{L}(D)$ is the language of good and bad prefixes of ϕ ,
- 2) D is the minimal deterministic finite automaton accepting the good and bad prefixes of ϕ .

Proof. 1) Suppose $s_1 s_2 \dots s_n$ be a good or bad prefix of ϕ . Then by Theorem 1, $GB(\phi\{s_1\}\{s_2\} \dots \{s_n\}) \in \{0, 1\}$. Let $\phi_i = \phi\{s_1\}\{s_2\} \dots \{s_i\}$; then $\phi_{i+1} = \phi_i\{a_{i+1}\}$. To prove that $s_1 s_2 \dots s_n \in \mathcal{L}(D)$, we use induction to show that for each $1 \leq i \leq n$, $\phi_i \equiv_{GB} \delta(\phi, s_1 s_2 \dots s_i)$. For the base case if $\phi_1 \equiv_{GB} \phi\{s_1\}$ then **dfs** extends the transition function by setting $\delta(\phi, s_1) = \phi$. Therefore, $\phi_1 \equiv_{GB} \phi = \delta(\phi, s_1)$. If $\phi_1 \not\equiv_{GB} \phi$ then **dfs** extends δ by setting $\delta(\phi, s_1) = \phi_1$. So $\phi_1 \equiv_{GB} \delta(\phi, s_1)$ holds in this case also. For the induction step let us assume that $\phi_i \equiv_{GB} \phi' = \delta(\phi, s_1 s_2 \dots s_i)$. If $\delta(\phi', s_{i+1}) = \phi''$ then from the **dfs** procedure we can see

that $\phi'' \equiv_{GB} \phi'\{s_{i+1}\}$. However, $\phi_i\{s_{i+1}\} \equiv_{GB} \phi'\{s_{i+1}\}$, since $\phi_i \equiv_{GB} \phi'$ by induction hypothesis. So $\phi_{i+1} \equiv_{GB} \phi'' = \delta(\phi', s_{i+1}) = \delta(\phi, s_1 s_2 \dots s_{i+1})$. Also notice $GB(\phi_n \equiv_{GB} \delta(\phi, s_1 s_2 \dots s_n)) \in \{0, 1\}$; this implies that $\delta(\phi, s_1 s_2 \dots s_n)$ is a final state and hence $s_1 s_2 \dots s_n \in \mathcal{L}(D)$.

Now suppose $s_1 s_2 \dots s_n \in \mathcal{L}(D)$. The proof that $s_1 s_2 \dots s_n$ is a good or bad prefix of ϕ goes in a similar way by showing that $\phi_i \equiv_{GB} \delta(\phi, s_1 s_2 \dots s_i)$.

2) If the automaton D is not minimal then there exists at least two states p and q in D such that p and q are equivalent [12] i.e. $\forall w \in \sigma^* : \delta(p, w) \in F$ if and only if $\delta(q, w) \in F$, where F is the set of final states. This means, if ϕ_1 and ϕ_2 are the LTL formulae associated with p and q respectively in **dfs** then $\phi_1 \equiv_{GB} \phi_2$. But **dfs** ensures that no two LTL formulae representing the states of the automaton are good-bad prefix equivalent. So we get a contradiction. \square

The GB-automaton thus generated can be used as a monitor for the given LTL formula. If at any point of monitoring we reach the state true in the GB-automaton we say that the monitored finite trace satisfies the LTL formula. If we reach the state false we say that the monitored trace violates the LTL formula. If we get stuck at some state i.e. we cannot take a transition, we say that the monitored trace is not monitorable. Otherwise we continue monitoring by consuming another state of the trace.

In the above procedure if we use the specification $\mathcal{B}_{LTL/B}$ (or $\mathcal{B}_{LTL/G}$) instead of $\mathcal{B}_{LTL/GB}$ and consider false (or true) as the only final state, we get a B-automaton (or G-automaton). These automata can detect either bad or good prefixes. Since the final state is either false or true the procedure to remove redundant states will result in smaller automata compared to the corresponding GB-automaton.

We have an implementation of the algorithm adapted to extended regular expressions which is available for evaluation on the internet via a CGI server reachable from <http://fsl.cs.uiuc.edu/rv/>.

5 Time and Space Complexity

Any possible derivative of an LTL formula ϕ , in its normal form, is an *exclusive or* of *conjunctions* of temporal subformulae (subformulae having temporal operators at the top) in ϕ . The number of such temporal subformulae is $O(m)$, where m is the size of ϕ . Hence, by counting argument, the number of possible conjuncts is $O(2^m)$. The number of possible exclusive ors of these conjuncts is then $O(2^{2^m})$. Therefore, the number of possible distinct derivatives of ϕ is $O(2^{2^m})$. Since the number states of the GB-automaton accepting good and bad prefixes of ϕ cannot be greater than the number of derivatives, 2^{2^m} is an upper bound on the number of possible states of the GB-automaton. Hence, the size of the GB-automaton is $O(2^{2^m})$. Thus we get the following lemma:

Lemma 1. *The size of the minimal GB-automaton accepting the good and bad prefixes of any LTL formula of size m is $O(2^{2^m})$.*

For the lower bound on the size of the automata we consider the language

$$L_k = \{\sigma \# w \# \sigma' \$ w \mid w \in \{0, 1\}^k \text{ and } \sigma, \sigma' \in \{0, 1, \#\}^*\}.$$

This language was previously used in several works [15, 16, 23] to prove lower bounds. The language can be expressed by the LTL formula [16] of size $O(k^2)$:

$$\phi_k = [(\neg \$) \mathcal{U} (\$ \mathcal{U} \square (\neg \$))] \wedge \Diamond [\# \wedge \bigwedge_{i=1}^n ((\circ^i 0 \wedge \square (\$ \rightarrow \circ^i 0)) \vee (\circ^i 1 \wedge \square (\$ \rightarrow \circ^i 1)))].$$

For this LTL formula the following result holds.

Lemma 2. *Any GB-automaton accepting good and bad prefixes of ϕ_k will have size $\Omega(2^{2^k})$.*

Proof: In order to prove the lower bound, the following equivalence relation on strings over $(0 + 1 + \#)^*$ is useful. For a string $\sigma \in (0 + 1 + \#)^*$, define $S(\sigma) = \{w \in (0 + 1)^k \mid \exists \lambda_1, \lambda_2. \lambda_1 \# w \# \lambda_2 = \sigma\}$. We will say that $\sigma_1 \equiv_k \sigma_2$ iff $S(\sigma_1) = S(\sigma_2)$. Now observe that the number of equivalence classes of \equiv_k is 2^{2^k} ; this is because for any $S \subseteq (0 + 1)^k$, there is a σ such that $S(\sigma) = S$.

We will prove this lower bound by contradiction. Suppose A is a GB-automaton that has a number of states less than 2^{2^k} for the LTL formula ϕ_k . Since the number of equivalence classes of \equiv_k is 2^{2^k} , by pigeon hole principle, there must be two strings $\sigma_1 \not\equiv_k \sigma_2$ such that the state of A after reading $\sigma_1 \$$ is the same as the state after reading $\sigma_2 \$$. In other words, A will reach the same state after reading inputs of the form $\sigma_1 \$ w$ and $\sigma_2 \$ w$. Now since $\sigma_1 \not\equiv_k \sigma_2$, it follows that $(S(\sigma_1) \setminus S(\sigma_2) \cup (S(\sigma_2) \setminus S(\sigma_1))) \neq \emptyset$. Take $w \in (S(\sigma_1) \setminus S(\sigma_2) \cup (S(\sigma_2) \setminus S(\sigma_1)))$. Then clearly, exactly one out of $\sigma_1 \$ w$ and $\sigma_2 \$ w$ is in L_k , and so A gives the wrong answer on one of these inputs. Therefore, A is not a correct GB-automaton. \square

Combining the above two results we get the following theorem.

Theorem 6. *The size of the minimal GB-automaton accepting the good and bad prefixes of any LTL formula of size m is $O(2^{2^m})$ and $\Omega(2^{2^{\sqrt{m}}})$.*

The space and time complexity of the algorithm is given by the following:

Theorem 7. *The LTL to optimal monitor generation algorithm requires $2^{O(2^m)}$ space and $c2^{O(2^m)}$ time for some constant c .*

Proof: The number of distinct derivatives of an LTL formula of size m can be $O(2^{2^m})$. Each such derivative can be encoded in space $O(2^m)$. So the number of circularities that are generated in the algorithm can consume $O(2^{2^m} 2^m 2^m)$ space. The space required by the algorithm is thus $2^{O(2^m)}$. \square

The number of iterations that the algorithm makes is less than the number of distinct derivatives. In each iteration the algorithm generates a set of circularities that can be at most $2^{O(2^m)}$. So the total time taken by the algorithm is $c2^{O(2^m)}$ for some constant c .

6 Conclusion and Future Work

In this paper we give a behavioral specification for LTL, which has the appealing property that two LTL formulae are *equivalent with respect to monitoring* if and

only if they are indistinguishable under carefully chosen experiments. To our knowledge, this is the first coalgebraic formalization of LTL. The major benefit of this formalization is that one can use *coinduction* to prove LTL formulae monitoring-equivalent, which can further be used to generate optimal LTL monitors on a single go. As future work we want to apply our coinductive techniques to generate monitors for other logics.

7 Acknowledgements

The work is supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586, the DARPA IXO NEST Program, contract number F33615-01-C-1907), the ONR Grant N00014-02-1-0715, the Motorola Grant MOTOROLA RPS #23 ANT, and the joint NSF/NASA grant CCR-0234524. We would like to thank Klaus Havelund and Predrag Tomic for reading a previous version of this paper and giving us valuable feedback and Howard Barringer for very helpful discussions, valuable feedback on the paper and pointers to references.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4), 1985.
2. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*. Springer, 2000.
3. M. Geilen. On the construction of monitors for temporal logic properties. In *ENTCS*, volume 55. Elsevier, 2001.
4. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*. IEEE, 2001. Coronado Island, California.
5. J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'00)*. IEEE, 2000. (Grenoble, France).
6. J. Goguen, K. Lin, and G. Rosu. Conditional circular coinductive rewriting with case analysis. In *Recent Trends in Algebraic Development Techniques (WADT'02)*, LNCS, Frauenschiemsee, Germany, September 2002. Springer.
7. K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), Apr. 2000.
8. K. Havelund and G. Roşu. Java PathExplorer – A Runtime Verification Tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, Montreal, Canada, June 18 - 21, 2001.
9. K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*. IEEE, 2001. Coronado Island, California.
10. K. Havelund and G. Roşu. *Runtime Verification 2002*, volume 70(4) of *ENTCS*. Elsevier, 2002. Proceedings of a *Computer Aided Verification (CAV'02)* satellite workshop.

11. K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*. Springer, 2002.
12. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
13. J. Hsiang. Refutational theorem proving using term rewriting systems. *Artificial Intelligence*, 25, 1985.
14. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier, 2001.
15. O. Kupferman and M. Y. Vardi. Freedom, Weakness, and Determinism: From linear-time to branching-time. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS'98)*, 1998.
16. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Proceedings of the Conference on Computer-Aided Verification (CAV'99)*, 1999.
17. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
18. Z. Manna, N. Bjørner, and A. B. et al. An update on STeP: Deductive-algorithmic verification of reactive systems. In *Tool Support for System Specification, Development and Verification*, LNCS. Springer, 1998.
19. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer-Verlag N.Y., Inc., 1995.
20. T. O'Malley, D. Richardson, and L. Dillon. Efficient Specification-Based Oracles for Critical Systems. In *Proceedings of the California Software Symposium*, 1996.
21. D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-Based Test Oracles for Reactive Systems. In *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE'92)*, 1992.
22. G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.
23. G. Roşu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In *Rewriting Techniques and Applications (RTA'03)*, LNCS. Springer, 2003.
24. K. Sen and G. Roşu. Generating Optimal Monitors for Extended Regular Expressions. In *Proceedings of Runtime Verification (RV'03) (To appear)*, volume 89(2) of *ENTCS*. Elsevier, 2003.
25. K. Sen, G. Roşu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '03)*, Helsinki, Finland, 2003.
26. S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*. Springer, 2000.
27. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings, The 15th IEEE International Conference on Automated Software Engineering (ASE'00)*. IEEE CS Press, Sept. 2000.

Appendix AG:

An Executable Specification of Asynchronous Pi-Calculus Semantics and May Testing in Maude 2.0

Prasanna Thati Koushik Sen

*Department of Computer Science
University of Illinois at Urbana-Champaign
{thati,ksen}@cs.uiuc.edu*

Narciso Martí-Oliet

*Dpto. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
narciso@sip.ucm.es*

Abstract

We describe an executable specification of the operational semantics of an asynchronous version of the π -calculus in Maude by means of conditional rewrite rules with rewrites in the conditions. We also present an executable specification of the may testing equivalence on non-recursive asynchronous π -calculus processes, using the Maude metalevel. Specifically, we describe our use of the `metaSearch` operation to both calculate the set of all finite traces of a non-recursive process, and to compare the trace sets of two processes according to a preorder relation that characterizes may testing in asynchronous π -calculus. Thus, in both the specification of the operational semantics and the may testing, we make heavy use of new features introduced in version 2.0 of the Maude language and system.

Key words: π -calculus, asynchrony, may testing, traces, Maude.

1 Introduction

Since its introduction in the seminal paper [11] by Milner, Parrow, and Walker, the π -calculus has become one of the most studied calculus for name-based mobility of processes, where processes are able to exchange names over channels so that the communication topology can change during the computation. The operational semantics of the π -calculus has been defined for several different versions of the calculus following two main styles. The first is the labelled transition system style according to the SOS approach introduced by Plotkin

©2003 Published by Elsevier Science B. V.

[13]. The second is the reduction style, where first an equivalence is imposed on syntactic processes (typically to make syntax more abstract with respect to properties of associativity and/or commutativity of some operators), and then some reduction or rewrite rules express how the computation proceeds by communication between processes.

The first specification of the π -calculus operational semantics in rewriting logic was developed by Viry in [19], in a reduction style making use of de Bruijn indexes, explicit substitutions, and reduction strategies in Elan [6]. This presentation was later improved by Stehr [14] by making use of a generic calculus for explicit substitutions, known as *CINNI*, which combines the best of the approaches based on standard variables and de Bruijn indices, and that has been implemented in Maude.

Our work took the work described above as a starting point, together with recent work by Verdejo and Martí-Oliet [18] showing how to use the new features of Maude 2.0 in the implementation of a semantics in the labelled transition system style for CCS. This work makes essential use of conditional rewrite rules with rewrites in the conditions, so that an inference rule in the labelled transition system of the form

$$\frac{P_1 \rightarrow Q_1 \quad \dots \quad P_n \rightarrow Q_n}{P_0 \rightarrow Q_0}$$

becomes a *conditional* rewrite rule of the form

$$P_0 \longrightarrow Q_0 \quad \text{if} \quad P_1 \longrightarrow Q_1 \wedge \dots \wedge P_n \longrightarrow Q_n,$$

where the condition includes rewrites. These rules are executable in version 2.0 of the Maude language and system [7]. However, this is not enough, because it is necessary to have some control on the application of rules. Typically, rewrite rules can be applied anywhere in a term, while the transitions in the operational semantics for CCS or the π -calculus in the SOS style only take place at the top. The new **frozen** attribute available in Maude 2.0 makes this possible, because the declaration of an operator as frozen forbids rewriting its arguments, thus providing another way of controlling the rewriting process. Rewrite conditions when applying conditional rules are solved by means of an implicit *search* process, which is also available to the user both at the command level and at the metalevel. The **search** command looks for all the rewrites of a given term that match a given pattern satisfying some condition. Search is reified at the metalevel as an operation **metaSearch**.

In this way, our first contribution is a fully executable specification of an operational semantics in the labelled transition system style for an asynchronous version of the π -calculus (the semantics for the synchronous case is obtained as a simple modification). This specification uses conditional rewrite rules with rewrites in conditions and the CINNI calculus [14] for managing names and bindings in the π -calculus. However, these two ingredients are not enough to obtain a fully executable specification. A central problem to overcome is that the transitions of a term can be *infinitely branching*. For instance,

the term $x(y).P$ can evolve via an input action to one of an infinite family of terms depending on the name received in the input at channel x . Our solution is to define the transitions of a process relative to an execution environment. The environment is represented abstractly as a set of free (global) names that the environment may use while interacting with the process, and transitions are modelled as rewrite rules over a pair consisting of a set of environment names together with a process.

Our next contribution is to implement the verification of the *may-testing preorder* [12,3,5] between finitary (non-recursive) asynchronous π -calculus processes, using again ideas from [18] to calculate the set of all finite traces of a process. May testing is a specific instance of the notion of behavioral equivalence on π -calculus processes; in may testing, two processes are said to be equivalent if they have the same success properties in all experiments. An experiment consists of an observing process that runs in parallel and interacts with the process being tested, and success is defined as the observer signalling a special event. Viewing the occurrence of an event as something bad happening, may testing can be used to reason about safety properties [4].

Since the definition of may testing involves a universal quantification over all observers, it is difficult to establish process equivalences directly from the definition. As a solution, alternate characterizations of the equivalence that do not resort to quantification over observers have been found. It is known that the trace semantics is an alternate characterization of may testing in (synchronous) π -calculus [3], while a variant of the trace semantics has been shown to characterize may testing in an asynchronous setting [5]. Specifically, in both these cases, comparing two processes according to the may-testing preorder amounts to comparing the set of all finite traces they exhibit. We have implemented for finite asynchronous processes, the comparison of trace sets proposed in [5]. We stress that our choice of specifying an asynchronous version rather than the synchronous π -calculus, is because the characterization of may testing for the asynchronous case is more interesting and difficult. The synchronous version can be specified in an executable way using similar but simpler techniques.

Our first step in obtaining an executable specification of may testing is to obtain the set of all finite traces of a given process. This is done at the Maude metalevel by using the `metaSearch` operation to collect all results of rewriting a given term. The second step is to specify a preorder relation between traces that characterizes may testing. We have represented the trace preorder relation as a rewriting relation, i.e. the rules of inference that define the trace preorder are again modeled as conditional rewrite rules. The final step is to check if two processes are related by the may preorder, i.e. whether a statement of the form $P \sqsubseteq Q$ is true or not. This step involves computing the closure of a trace under the trace-preorder relation, again by means of the `metaSearch` operation. Thus, our work demonstrates the utility of the new metalevel facilities available in Maude 2.0.

The structure of the paper follows the steps in the description above. Section 2 describes the syntax of the asynchronous version of the π -calculus that we consider, together with the corresponding CINNI operations we use. Section 3 describes the operational semantics specified by means of conditional rewrite rules. Sections 4 and 5 define traces and the preorder on traces, respectively. Finally, Section 6 contains the specification of the may testing on processes as described above. Section 7 concludes the paper along with a brief discussion of future work.

Although this paper includes some information on the π -calculus and may testing to make it as self contained as possible, we refer the reader to the papers [5,3,11] for complete details on these subjects. In the same way, the interested reader can find a detailed explanation about the new features of Maude 2.0 in [7], and about their use in the implementation of operational semantics in the companion paper [18].

2 Asynchronous π -Calculus Syntax

The following is a brief and informal review of a version of asynchronous π -calculus that is equipped with a conditional construct for matching names. An infinite set of channel names is assumed, and u, v, w, x, y, z, \dots are assumed to range over it. The set of processes, ranged over by P, Q, R , is defined by the following grammar:

$$P := \bar{x}y \mid \sum_{i \in I} \alpha_i.P_i \mid P_1|P_2 \mid (\nu x)P \mid [x = y](P_1, P_2) \mid !P$$

where α can be $x(y)$ or τ .

The output term $\bar{x}y$ denotes an asynchronous message with target x and content y . The summation $\sum_{i \in I} \alpha_i.P_i$ non-deterministically chooses an α_i , and if $\alpha_i = \tau$ it evolves internally to P_i , and if $\alpha_i = x(y)$ it receives an arbitrary name z at channel x and then behaves like $P\{z/y\}$. The process $P\{z/y\}$ is the result of the substitution of free occurrences of y in P by z , with the usual renaming of bound names to avoid accidental captures (thus substitution is defined only modulo α -equivalence). The argument y in $x(y).P$ binds all free occurrences of y in P . The composition $P_1|P_2$ consists of P_1 and P_2 acting in parallel. The components can act independently, and also interact with each other. The restriction $(\nu x)P$ behaves like P except that it can not exchange messages targeted to x , with its environment. The restriction binds free occurrences of x in P . The conditional $[x = y](P_1, P_2)$ behaves like P_1 if x and y are identical, and like P_2 otherwise. The replication $!P$ provides an infinite number of copies of P . The functions for free names $fn(\cdot)$, bound names $bn(\cdot)$ and names $n(\cdot)$, of a process, are defined as expected.

In the Maude specification for the π -calculus syntax that follows, the sort `Chan` is used to represent channel names and each of the non-constant syntax constructors is declared as `frozen`, so that the corresponding arguments

cannot be rewritten by rules; this will be justified at the end of Section 3.

```

sort Chan .
sorts Guard GuardedTrm SumTrm Trm .
subsort GuardedTrm < SumTrm .
subsort SumTrm < Trm .

op _(_ ) : Chan Qid -> Guard .
op tau : -> Guard .
op nil : -> Trm .
op _<_> : Chan Chan -> Trm [frozen] .
op _.._ : Guard Trm -> GuardedTrm [frozen] .
op _+_ : SumTrm SumTrm -> SumTrm [frozen assoc comm] .
op _|_ : Trm Trm -> Trm [frozen assoc comm] .
op new[_]_ : Qid Trm -> Trm [frozen] .
op if_=then_else_fi : Chan Chan Trm Trm -> Trm [frozen] .
op !_ : Trm -> Trm [frozen] .

```

Note that the syntactic form $\sum_{i \in I} \alpha_i.P_i$ has been split into three cases:

- (i) `nil` represents the case where $I = \emptyset$,
- (ii) a term of sort `GuardedTrm` represents the case where $I = \{1\}$, and
- (iii) a term of sort `SumTrm` represents the case where $I = [1..n]$ for $n > 1$. Since the constructor `_+_` is associative and the sort `GuardedTrm` is a subsort of `SumTrm`, we can represent a finite sum $\sum_{i \in I} \alpha_i.P_i$ as $(\dots (\alpha_1.P_1 + \alpha_2.P_2) + \dots \alpha_n.P_n)$.

To represent substitution on π -calculus processes (and traces, see Section 4) at the language level we use CINNI as a calculus for explicit substitutions [14]. This gives a first-order representation of terms with bindings and capture-free substitutions, instead of going to the metalevel to handle names and bindings. The main idea in such a representation is to keep the bound names inside the binders as it is, but to replace its use by the name followed by an index which is a count of the number of binders with the same name it jumps before it reaches the place of use. Following this idea, we define terms of sort `Chan` as indexed names as follows.

```

sort Chan .
op {_} : Qid Nat -> Chan [prec 1] .

```

We introduce a sort of substitutions `Subst` together with the following operations:

```

op [_:=_] : Qid Chan -> Subst .
op [shiftup_] : Qid -> Subst .
op [shiftdown_] : Qid -> Subst .
op [lift_] : Qid Subst -> Subst .

```

The first two substitutions are basic substitutions representing *simple* and *shiftup* substitutions; the third substitution is a special case of *simple* substitution; the last one represents complex substitution where a substitution can be lifted using the operator `lift`. The intuitive meaning of these operations

[a := x]	[shiftup a]	[shiftdown a]	[lift a S]
a{0} ↪ x	a{0} ↪ a{1}	a{0} ↪ a{0}	a{0} ↪ [shiftup a] (S a{0})
a{1} ↪ a{0}	a{1} ↪ a{2}	a{1} ↪ a{0}	a{1} ↪ [shiftup a] (S a{1})
...
a{n+1} ↪ a{n}	a{n} ↪ a{n+1}	a{n+1} ↪ a{n}	a{n} ↪ [shiftup a] (S a{n})
b{m} ↪ b{m}	b{m} ↪ b{m}	b{m} ↪ b{m}	b{m} ↪ [shiftup a] (S b{m})

Table 1
The CINNI operations.

is described in Table 1 (see [14] for more details). Using these, explicit substitutions for π -calculus processes are defined equationally. Some interesting equations are the following:

$$\begin{aligned} \text{eq } & S(P + Q) = (S P) + (S Q) . \\ \text{eq } & S(CX(Y) . P) = (S CX)(Y) . ([lift Y S] P) . \\ \text{eq } & S(\text{new } [X] P) = \text{new } [X] ([lift X S] P) . \end{aligned}$$

3 Operational Semantics

A labelled transition system (see Table 2) is used to give an operational semantics for the calculus as in [5]. The transition system is defined modulo α -equivalence on processes in that α -equivalent processes have the same transitions. The rules *COM*, *CLOSE*, and *PAR* have symmetric versions that are not shown in the table.

Transition labels, which are also called *actions*, can be of five forms: τ (a silent action), $\bar{x}y$ (free output of a message with target x and content y), $\bar{x}(y)$ (bound output), xy (free input of a message), and $x(y)$ (bound input). The functions $fn(.)$, $bn(.)$ and $n(.)$ are defined on actions as expected. The set of all visible (non- τ) actions is denoted by \mathcal{L} , and α is assumed to range over \mathcal{L} . As a uniform notation for free and bound actions the following notational convention is adopted: $(\emptyset)\bar{x}y = \bar{x}y$, $(\{y\})\bar{x}y = \bar{x}(y)$, and similarly for input actions. The variable \hat{z} is assumed to range over $\{\emptyset, \{z\}\}$. The term $(\nu z)P$ is $(\nu z)P$ if $\hat{z} = \{z\}$, and P otherwise.

We define the sort **Action** and the corresponding operations as follows:

```
sorts Action ActionType .
ops i o : -> ActionType .
op f : ActionType Chan Chan -> Action .
op b : ActionType Chan Qid -> Action .
op tauAct : -> Action .
```

The operators **f** and **b** are used to construct free and bound actions respectively. Name substitution on actions is defined equationally as expected.

The inference rules in Table 2 are modelled as conditional rewrite rules

$INP: \sum_{i \in I} \alpha_i.P_i \xrightarrow{x_j z} P_j\{z/y\} \ j \in I, \alpha_j = x_j(y)$	$OUT: \bar{xy} \xrightarrow{\bar{xy}} 0$
$TAU: \sum_{i \in I} \alpha_i.P_i \xrightarrow{\tau} P_j \ j \in I, \alpha_j = \tau$	$BINP: \frac{P \xrightarrow{xy} P'}{P \xrightarrow{x(y)} P'} \ y \notin fn(P)$
$PAR: \frac{P_1 \xrightarrow{\alpha} P'_1}{P_1 P_2 \xrightarrow{\alpha} P'_1 P_2} \ bn(\alpha) \cap fn(P_2) = \emptyset$	$COM: \frac{P_1 \xrightarrow{\bar{xy}} P'_1 \ P_2 \xrightarrow{xy} P'_2}{P_1 P_2 \xrightarrow{\tau} P'_1 P'_2}$
$RES: \frac{P \xrightarrow{\alpha} P'}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'} \ y \notin n(\alpha)$	$OPEN: \frac{P \xrightarrow{\bar{xy}} P'}{(\nu y)P \xrightarrow{\bar{x}(y)} P'} \ x \neq y$
$CLOSE: \frac{P_1 \xrightarrow{\bar{x}(y)} P'_1 \ P_2 \xrightarrow{xy} P'_2}{P_1 P_2 \xrightarrow{\tau} (\nu y)(P'_1 P'_2)} \ y \notin fn(P_2)$	$REP: \frac{P !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$
$IF: \frac{P \xrightarrow{\alpha} P'}{[x = x](P, Q) \xrightarrow{\alpha} P'}$	$ELSE: \frac{Q \xrightarrow{\alpha} Q'}{[x = y](P, Q) \xrightarrow{\alpha} Q'} \ x \neq y$

Table 2
A labelled transition system for asynchronous π -calculus.

with the premises as conditions of the rule.¹ Since rewrites do not have labels unlike the labelled transitions, we make the label a part of the resulting term; thus rewrites corresponding to transitions in the operational semantics are of the form $P \Rightarrow \{\alpha\}Q$.

Because of the INP and $OPEN$ rules, the transitions of a term can be infinitely branching. Specifically, in case of the INP rule there is one branch for every possible name that can be received in the input. In case of the $OPEN$ rule, there is one branch for every name that is chosen to denote the private channel that is being emitted (note that the transition rules are defined only modulo α -equivalence). To overcome this problem, we define transitions over pairs of the form $[CS] \ P$, where CS is a set of channel names containing all the names that the environment with which the process interacts, knows about. The set CS expands during bound input and output interactions when private names are exchanged between the process and its environment.

The infinite branching due to the INP rule is avoided by allowing only the names in the environment set CS to be received in free inputs. Since CS is assumed to contain all the free names in the environment, an input argument that is not in CS would be a private name of the environment. Now, since the identifier chosen to denote the fresh name is irrelevant, all bound input

¹ The symmetric versions missing in the table need not be implemented because the process constructors $_+_$ and $_|__$ have been declared as commutative.

transitions can be identified to a single input. With these simplifications, the number of input transitions of a term become finite. Similarly, in the *OPEN* rule, since the identifier chosen to denote the private name emitted is irrelevant, instances of the rule that differ only in the chosen name are not distinguished.

We discuss in detail the implementation of only a few of the inference rules; the reader is referred to the appendix for a complete list of all the rewrite rules for Table 2.

```
sorts EnvTrm TraceTrm .
subsort EnvTrm < TraceTrm .
op  []_ : Chanset Trm -> EnvTrm [frozen] .
op  {}_ : Action TraceTrm -> TraceTrm [frozen] .
```

Note that the two operators are also declared above with the `frozen` attribute, forbidding in this way rewriting of their arguments, as justified at the end of this section.

The following non-conditional rule is for free inputs.

```
r1 [Inp] : [CY CS] ((CX(X) . P) + SUM) =>
           {f(i,CX,CY)} ([CY CS] ([X := CY] P)) .
```

The next rule we consider is the one for bound inputs. Since the identifier chosen to denote the bound argument is irrelevant, we use the constant '`U`' for all bound inputs, and thus '`U{0}`' denotes the fresh channel received. Note that in contrast to the *BINP* rule of Table 2, we do not check if '`U{0}`' is in the free names of the process performing the input, and instead we shift up the channel indices appropriately, in both the set of environment names `CS` and the process `P` in the righthand side and condition of the rule. This is justified because the transition target is within the scope of the bound name in the input action. Note also that the channel `CX` in the action is not shifted down because it is out of the scope of the bound argument. The set of environment names is expanded by adding the received channel '`U{0}`' to it. Finally, we use a special constant `flag` of sort `Chan`, to ensure termination. We add an instance of `flag` to the environment set of the rewrite in condition, so that the *BINP* rule is not fired again while evaluating the condition. Without this check, we will have a non-terminating execution in which the *BINP* rule is repeatedly fired.

```
crl [BInp] : [CS] P => {b(i,CX,'U)} ['U{0} [shiftup 'U] CS] P1
    if (not flag in CS) /\
        CS1 := flag 'U{0} [shiftup 'U] CS /\
        [CS1] [shiftup 'U] P => {f(i,CX,'U{0})} [CS1] P1 .
```

The following rule treats the case of bound outputs.

```
crl [Open] : [CS] (new [X] P) => {[shiftdown X] b(o,CY,X)} [X{0} CS1] P1
    if CS1 := [shiftup X] CS /\
        [CS1] P => {f(o,CY,X{0})} [CS1] P1 /\
        X{0} /= CY .
```

Like in the case of bound inputs, we identify all bound outputs to a single

instance in which the identifier X that appears in the restriction is chosen as the bound argument name. Note that in both the righthand side of the rule and in the condition, the indices of the channels in CS are shifted up, because they are effectively moved across the restriction. Similarly, the channel indices in the action in the righthand side of the rule are shifted down since the action is now moved out of the restriction. Note also that the exported name is added to the set of environment names, because the environment that receives this exported name can use it in subsequent interactions.

The PAR inference rule is implemented by two rewrite rules, one for the case where the performed action is free, and the other where the action is bound. The rewrite rule for the latter case is discussed next, while the one for the former case is simpler and appears in the appendix.

```
var IO : ActionType
crl [Par] : [CS] (P | Q) =>
    {b(IO,CX,Y)} [Y{0}] ([shiftup Y] CS) (P1 | [shiftup Y] Q)
    if [CS] P => {b(IO,CX,Y)} ([CS1] P1) .
```

Note that the side condition of the PAR rule in Table 2, which avoids confusion of the emitted bound name with free names in Q , is achieved by shifting up channel indices in Q . This is justified because the righthand side of the rule is under the scope of the bound output action. Similarly, the channel indices in the environment are also shifted up. Further, the set of environment names is expanded by adding the exported channel $Y\{0\}$.

Finally, we consider the rewrite rule for $CLOSE$. The process P emits a bound name Y , which is received by process Q . Since the scope of Y after the transition includes Q , the rewrite involving Q in the second condition of the rule is carried out within the scope of the bound name that is emitted. This is achieved by adding the channel $Y\{0\}$ to the set of environment names and shifting up the channel indices in both CS and Q in the rewrite. Note that since the private name being exchanged is not emitted to the environment, we neither expand the set CS in the righthand side of the rule nor shift up the channel indices in it.

```
crl [Close] : [CS] (P | Q) => {tauAct} [CS] new [Y] (P1 | Q1)
    if [CS] P => {b(o,CX,Y)} [CS1] P1 ∧
        [Y{0}] [shiftup Y] CS [shiftup Y] Q =>
            {f(i,CX,Y{0})} [CS2] Q1 .
```

We conclude this section with the following note. The operator $\{\cdot\}_-$ is declared **frozen** because further rewrites of the process term encapsulated in a term of sort **TraceTrm** are useless. This is because all the conditions of the transition rules only involve one step rewrites (the righthand side of these rewrites can only match a term of sort **TraceTrm** with a single action prefix). Further note that, to prevent rewrites of a term to a non well-formed term, all the constructors for π -calculus terms (Section 2) have been declared **frozen**; in the absence of this declaration we would have for instance rewrites of the form $P | Q \Rightarrow \{A\}.P1 | Q$ to a non well-formed term.

4 Trace Semantics

The set \mathcal{L}^* is the set of *traces*. The functions $fn(\cdot)$, $bn(\cdot)$ and $n(\cdot)$ are extended to \mathcal{L}^* in the obvious way. The relation of α -equivalence on traces is defined as expected, and α -equivalent traces are not distinguished. The relation \Rightarrow denotes the reflexive transitive closure of $\xrightarrow{\tau}$, and $\xrightarrow{\beta}$ denotes $\Rightarrow \xrightarrow{\beta} \Rightarrow$. For $s = l.s'$, we inductively define $P \xrightarrow{s} P'$ as $P \xrightarrow{l} \xrightarrow{s'} P'$. We use $P \xrightarrow{s}$ as an abbreviation for $P \xrightarrow{s} P'$ for some P' . The set of traces that a process exhibits is then $\|P\| = \{s \mid P \xrightarrow{s}\}$.

In the implementation, we introduce a sort **Trace** as supersort of **Action** to specify traces.

```
subsort Action < Trace .
op epsilon : -> Trace .
op _.._ : Trace Trace -> Trace [assoc id: epsilon] .
op [] : Trace -> TTrace .
```

We define the operator `[]` to represent a complete trace. The motivation for doing so is to restrict the equations and rewrite rules defined over traces to operate only on a complete trace instead of a part of it. The following equation defines α -equivalence on traces. Note that in a trace `TR1.b(I0,CX,Y).TR2` the action `b(I0,CX,Y)` binds the identifier `Y` in `TR2`.

```
ceq [TR1 . b(I0,CX,Y) . TR2] =
    [TR1 . b(I0,CX,'U) . [Y := 'U{0}] [shiftup 'U] TR2]
    if Y /= 'U .
```

Because the operator `op {}_- : Action TraceTrm -> TraceTrm` is declared as **frozen**, a term of sort **EnvTrm** can rewrite only once, and so we cannot obtain the set of finite traces of a process by simply rewriting it multiple times in all possible ways. The problem is solved as in [18], by specifying the trace semantics using rules that generate the transitive closure of one step transitions as follows:

```
sort TTrm .
op [] : EnvTrm -> TTrm [frozen] .
var TT : TraceTrm .

crl [reflx] : [ P ] => {A} Q if P => {A} Q .
crl [trans] : [ P ] => {A} TT
    if P => {A} Q /\ [ Q ] => TT /\ [ Q ] /= TT .
```

We use the operator `[]` to prevent infinite loops while evaluating the conditions of the rules above. If this operator were not used, then the lefthand side of the rewrite in the condition would match the lefthand side of the rule itself, and so the rule itself could be used in order to solve its condition. This operator is also declared as **frozen** to prevent useless rewrites inside `[]`.

We can now use the **search** command of Maude 2.0 to find all possible traces of a process. The traces appear as prefix of the one-step successors of a **TTrm** of the form `[[CS] P]`. For instance, the set of all traces exhibited

by $[\text{mt}] \text{ new } [y] (x_0 < y_0 > \mid x_0(u) . \text{nil})$ (where mt denotes the empty channel set), can be obtained by using the following `search` command.

```
Maude> search [ [mt] new [y] (x{0} < y{0} > | x{0}(u) . nil) ] =>!
X:TraceTrm .
search in APITRACESET : [[mt]new[y](x{0} < y{0} > | x{0}(u) . nil)] =>!
X:TraceTrm .

Solution 1 (state 1)
states: 7  rewrites: 17344 in 110ms cpu (150ms real) (157672 rewrites/second)
X:TraceTrm --> {b(i, 'x{0}, 'u)}['u{0}]new[y](nil | 'x{0} < 'y{0} >)

Solution 2 (state 2)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {tauAct}[mt]new[y](nil | nil)

Solution 3 (state 3)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(o, 'x{0}, 'y)}['y{0}]nil | 'x{0}(u) . nil

Solution 4 (state 4)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(i, 'x{0}, 'u)}{b(o, 'x{0}, 'y)}['y{0} 'u{0}]nil | nil

Solution 5 (state 5)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(o, 'x{0}, 'y)}{b(i, 'x{0}, 'u)}['y{0} 'u{0}]nil | nil

Solution 6 (state 6)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(o, 'x{0}, 'y)}{f(i, 'x{0}, 'y{0})}['y{0}]nil | nil

No more solutions.
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
```

The command returns all `TraceTrm`s that can be reached from the given `TTrm`, and that are terminating (the ‘!’ in $=>!$ specifies that the target should be terminating). The required set of traces can be obtained by simply extracting from each solution $\{a_1\} \dots \{a_n\} T T$ the sequence $a_1 \dots a_n$ and removing all `tauActs` in it. Thus, we have obtained an executable specification of the trace semantics of asynchronous π -calculus.

5 A Trace Based Characterization of May Testing

The may-testing framework [12] is instantiated on asynchronous π -calculus as follows. Observers are processes that can emit a special message $\bar{\mu}\mu$. We say that an observer O accepts a trace s if $O \xrightarrow{\bar{s}.\bar{\mu}\mu}$, where \bar{s} is the trace obtained by complementing the actions in s , i.e. converting input actions to output actions and vice versa. The may preorder \sqsubseteq over processes is defined as: $P \sqsubseteq Q$ if for every observer O , $P|O \xrightarrow{\bar{\mu}\mu}$ implies $Q|O \xrightarrow{\bar{\mu}\mu}$. We say that P and Q are *may-*

<i>(Drop)</i>	$s_1.(\hat{y})s_2 \prec s_1.(\hat{y})xy.s_2$ if $(\hat{y})s_2 \neq \perp$
<i>(Delay)</i>	$s_1.(\hat{y})(\alpha.xy.s_2) \prec s_1.(\hat{y})xy.\alpha.s_2$ if $(\hat{y})(\alpha.xy.s_2) \neq \perp$
<i>(Annihilate)</i>	$s_1.(\hat{y})s_2 \prec s_1.(\hat{y})xy.\bar{x}y.s_2$ if $(\hat{y})s_2 \neq \perp$

Table 3
A preorder relation on traces.

equivalent, i.e. $P = Q$, if $P \sqsubseteq Q$ and $Q \sqsubseteq P$. The universal quantification on contexts in this definition makes it very hard to prove equalities directly from the definition, and makes mechanical checking impossible. To circumvent this problem, a trace based alternate characterization of the may equivalence is proposed in [5]. We now summarize this characterization and discuss our implementation of it.

The preorder \preceq on traces is defined as the reflexive transitive closure of the laws shown in Table 3, where the notation $(\hat{y})\cdot$ is extended to traces as follows.

$$(\hat{y})s = \begin{cases} s & \text{if } \hat{y} = \emptyset \text{ or } b \notin fn(s) \\ s_1.x(y).s_2 & \text{if } \hat{y} = \{y\} \text{ and there are } s_1, s_2, x \text{ such that} \\ & s = s_1.xy.s_2 \text{ and } y \notin n(s_1) \cup \{x\} \\ \perp & \text{otherwise} \end{cases}$$

For sets of traces R and S , we define $R \precsim S$, if for every $s \in S$ there is an $r \in R$ such that $r \preceq s$. The may preorder is then characterized in [5] as: $P \sqsubseteq Q$ if and only if $\|Q\| \precsim \|P\|$.

The main intuition behind the preorder \preceq is that if an observer accepts a trace s , then it also accepts any trace $r \preceq s$. The first two laws state that an observer cannot force inputs on the process being tested. Since outputs are asynchronous, the actions following an output in a trace exhibited by the observer need not causally depend on the output. Hence the observer's output can be delayed until a causally dependent action, or dropped if there are no such actions. The annihilation law states that an observer can consume its own outputs unless there are subsequent actions that depend on the output. The reader is referred to [5] for further details on this characterization.

We encode the trace preorder as rewrite rules on terms of the sort **TTrace** of complete traces; specifically, the relation $r \prec s$ if cond, is encoded as **s => r if cond**. The reason for this form of representation will be justified in Section 6. The function $(\{y\})\cdot$ on traces is defined equationally by the operation **bind**. The constant **bot** of sort **Trace** is used by the bind operation to signal error.

```
op bind : Qid Trace -> Trace .
op bot  : -> Trace .
var TR  : Trace .    var IO  : ActionType.
```

```

ceq TR . bot = bot if t /= epsilon .
ceq bot . TR = bot if t /= epsilon .

eq bind(X , epsilon) = epsilon .

eq bind(X , f(i,CX,CY) . TR ) = if CX /= X{0} then
    if CY == X{0} then ([shiftdown X] b(i, CX , X)) . TR
    else ([shiftdown X] f(i, CX , CY)) . bind(X , TR) fi
    else bot fi .

eq bind(X , b(IO,CX,Y) . TR) = if CX /= X{0} then
    if X /= Y then ([shiftdown X] b(i, CX , Y)) . bind(X , TR)
    else ([shiftdown X] b(IO, CX , Y)) . bind(X , swap(X,TR)) fi
    else bot fi .

```

The equation for the case where the second argument to **bind** begins with a free output is not shown as it is similar. Note that the channel indices in actions until the first occurrence of $X\{0\}$ as the argument of a free input are shifted down as these move out of the scope of the binder X . Further, when a bound action with X as the bound argument is encountered, the **swap** operation is applied to the remaining suffix of the trace. The swap operation simply changes the channel indices in the suffix so that the binding relation is unchanged even as the binder X is moved across the bound action. This is accomplished by simultaneously substituting $X\{0\}$ with $X\{1\}$, and $X\{1\}$ with $X\{0\}$. Finally, note that when $X\{0\}$ is encountered as the argument of a free input, the input is converted to a bound input. If $X\{0\}$ is first encountered at any other place, an error is signalled by returning the constant **bot**.

The encoding of the preorder relation on traces is now straightforward.

```

crl [Drop] : [ TR1 . b(i,CX,Y) . TR2 ] => [ TR1 . bind(Y , TR2) ]
    if bind(Y , TR2) /= bot .

rl [Delay] : [ ( TR1 . f(i,CX,CY) . b(IO,CU,V) . TR2 ) ] =>
    [ ( TR1 . b(IO,CU,V) . ([shiftup V] f(i, CX , CY)) . TR2 ) ] .

crl [Delay] : [ ( TR1 . b(i,CX,Y) . f(IO,CU,CV) . TR2 ) ] =>
    [ ( TR1 . bind(Y , f(IO,CU,CV) . f(i,CX,Y{0}) . TR2) ) ]
    if bind(Y , f(IO,CU,CV) . f(i,CX,Y{0}) . TR2) /= bot .

crl [Annihilate] : [ ( TR1 . b(i,CX,Y) . f(o,CX,Y{0}) . TR2 ) ] =>
    [ TR1 . bind(Y , TR2) ]
    if bind(Y , TR2) /= bot .

```

Note that in the first **Delay** rule, the channel indices of the free input action are shifted up when it is delayed across a bound action, since it gets into the scope of the bound argument. Similarly, in the second **Delay** rule, when the bound input action is delayed across a free input/output action, the channel indices of the free action are shifted down by the **bind** operation. The other two subcases of the **Delay** rule, namely, where a free input is to be delayed across a free input or output, and where a bound input is to be delayed across a bound input or output, are not shown as they are similar.

Similarly, for `Annihilate`, the case where a free input is to be annihilated with a free output is not shown.

6 Verifying the May Preorder between Finite Processes

We now describe our implementation of verification of the may preorder between finite processes, i.e. processes without replication, by exploiting the trace-based characterization of may testing discussed in Section 5. The finiteness of a process P only implies that the length of traces in $\|P\|$ is bounded, but the number of traces in $\|P\|$ can be infinite (even modulo α -equivalence) because the *INP* rule is infinitely branching. To avoid the problem of having to compare infinite sets, we observe that

$$\|Q\| \precsim \|P\| \quad \text{if and only if} \quad \|Q\|_{fn(P,Q)} \precsim \|P\|_{fn(P,Q)},$$

where for a set of traces S and a set of names ρ we define $S_\rho = \{s \in S \mid fn(s) \subseteq \rho\}$. Now, since the traces in $\|P\|$ and $\|Q\|$ are finite in length, it follows that the sets of traces $\|P\|_{fn(P,Q)}$ and $\|Q\|_{fn(P,Q)}$ are finite modulo α -equivalence. In fact, the set of traces generated for $[[fn(P,Q)] P]$ by our implementation described in Section 3, contains exactly one representative from each α -equivalence class of $\|P\|_{fn(P,Q)}$.

Given processes P and Q , we generate the set of all traces (modulo α -equivalence) of $[[fn(P,Q)] P]$ and $[[fn(P,Q)] Q]$ using the metalevel facilities of Maude 2.0. As mentioned in Section 4, these terms, which are of sort `TTrm`, can be rewritten only once. The term of sort `TraceTrm` obtained by rewriting contains a finite trace as a prefix. To create the set of all traces, we compute all possible one-step rewrites. This computation is done at the metalevel by the function `TTrmtoNormalTraceSet` that uses two auxiliary functions `TTrmtoTraceSet` and `TraceSettoNormalTraceSet`.

```
op TTrmtoTraceSet : Term -> TermSet .
op TraceSettoNormalTraceSet : TermSet -> TermSet .
op TTrmtoNormalTraceSet : Term -> TermSet .

eq TTrmtoNormalTraceSet(T) = TraceSettoNormalTraceSet(TTrmtoTraceSet(T)) .
```

The function `TTrmTraceSet` uses the function `allOneStepAux(T,N)` that returns the set of all one-step rewrites (according to the rules in Sections 3 and 4, which are defined in modules named `APISEMATICS` and `APITRACE`, see Figure A.1 in appendix) of the term T which is the metarepresentation of a term of sort `TTrm`, skipping the first N solutions. In the following equations, the operator `_u_` stands for set union.

Notice the use of the operation `metaSearch`, which receives as arguments the metarepresented module to work in, the starting term for search, the pattern to search for, a side condition (empty in this case), the kind of search (which may be `'*` for zero or more rewrites, `'+` for one or more rewrites, and `'!` for only matching normal forms), the depth of search, and the required solution number. It returns the term matching the pattern, its type, and

the substitution produced by the match; to keep only the term, we use the projection `getTerm`.

```

op APITRACE-MOD : -> Module .
eq APITRACE-MOD = ['APITRACE] .
var N : MachineInt .      vars T X : Term .

op allOneStepAux : Term MachineInt Term -> TermSet .
op TraceTermToTrace : Term -> Term .

eq TTrmToTraceSet(T) = allOneStepAux(T,0,'X:TraceTrm) .
eq allOneStepAux(T,N,X) =
    if metaSearch(APITRACE-MOD,T,X,nil,'+,1,N) == failure
    then 'epsilon.Trace
    else TraceTermToTrace(getTerm(metaSearch(APITRACE-MOD,T,X,nil,'+,1,N)))
        u allOneStepAux(T,N + 1,X) fi .

```

The function `TraceTrmToTrace` (whose equations are not shown), used in `allOneStepAux`, extracts the trace $a_1.a_2\dots a_n$ out of a metarepresentation of a term of sort `TraceTrm` of the form $\{a_1\}\{a_2\}\dots\{a_n\}TT$. The function `TraceSettoNormalTraceSet` uses the metalevel operation `metaReduce` to convert each trace in a trace set to its α -normal form. The operation `metaReduce` takes as arguments a metarepresented module and a metarepresented term in that module, and returns the metarepresentation of the fully reduced form of the given term using the equations in the given module, together with its corresponding sort or kind. Again, the projection `getTerm` leaves only the resulting term.

```

eq TraceSettoNormalTraceSet(mt) = mt .
eq TraceSettoNormalTraceSet(T u TS) =
    getTerm(metaReduce(TRACE-MOD,'[ _ ] [ T ]))
    u TraceSettoNormalTraceSet(TS) .

```

We implement the relation \precsim on sets defined in Section 5 as the predicate \ll . We check if $P \sqsubseteq Q$ by computing this predicate on the metarepresented trace sets $\|P\|_{fn(P,Q)}$ and $\|Q\|_{fn(P,Q)}$ as follows. For each (metarepresented) trace T in $\|P\|_{fn(P,Q)}$, we compute the reflexive transitive closure of T with respect to the laws shown in Table 3. The laws are implemented as rewrite rules in the module `TRACE-PREORDER`. We then use the fact that $\|Q\|_{fn(P,Q)} \precsim \|P\|_{fn(P,Q)}$ if and only if for every trace T in $\|P\|_{fn(P,Q)}$ the closure of T and $\|Q\|_{fn(P,Q)}$ have a common element.

```

op TRACE-PREORDER-MOD : -> Module .
eq TRACE-PREORDER-MOD = ['TRACE-PREORDER] .
var N : MachineInt .      vars T T1 T2 X : Term .
var TS TS1 TS2 : TermSet .

op _<<_ : TermSet TermSet -> Bool .
op _<<<_ : TermSet Term -> Bool .
op TTraceClosure : Term -> TermSet .
op TTraceClosureAux : Term Term MachineInt -> TermSet .
op _maypre_ : Term Term -> Bool .

```

```

eq TS2 << mt = true .
eq TS2 << (T1 u TS1) = TS2 <<< T1 and TS2 << TS1 .
eq TS2 <<< T1 = not disjoint?(TS2 , TTraceClosure(T1)) .
eq T1 maypre T2 = TTrmtoNormalTraceSet(T2) << TTrmtoNormalTraceSet(T1) .

```

The computation of the closure of T is done by the function `TTraceClosure`. It uses `TTraceClosureAux` to compute all possible (multi-step) rewrites of the term T using the rules defined in the module `TRACE-PREORDER`, again by means of the metalevel operation `metaSearch`.

```

eq TTraceClosure(T) = TTraceClosureAux(T,'TT:TTrace,0) .
eq TTraceClosureAux(T,X,N) =
  if metaSearch(TRACE-PREORDER-MOD,T,X,nil,'*,maxMachineInt,N) == failure
  then mt
  else getTerm(metaSearch(TRACE-PREORDER-MOD,T,X,nil,'*,maxMachineInt,N))
    u TTraceClosureAux(T,X,N + 1) fi .

```

This computation is terminating as the number of traces to which a trace can rewrite using the trace preorder laws is finite modulo α -equivalence. This follows from the fact that the length of a trace is non-increasing across rewrites, and the free names in the target of a rewrite are also free names in the source. Since the closure of a trace is finite, `metaSearch` can be used to enumerate all the traces in the closure. Note that although the closure of a trace is finite, it is possible to have an infinite rewrite that loops within a subset of the closure. Further, since T is a metarepresentation of a trace, `metaSearch` can be applied directly to T inside the function `TTraceClosureAux(T,X,N)`.

We end this section with a small example, which checks for the may-testing preorder between the processes $P = a(u).b(v).(\nu w)(\bar{w}v|\bar{a}u) + b(u).a(v).(\bar{b}u|\bar{b}w)$ and $Q = b(u).(\bar{b}u|\bar{b}w)$. We define constants `TP` and `TQ` of sort `TTrm`, along with the following equations:

```

eq TP = [[{'a{0}' 'b{0}' 'w{0}']
           'a{0}('u) . 'b{0}('v) . new['w]('w{0} < 'v{0} > | 'a{0} < 'u{0} >)
           + 'b{0}('u) . 'a{0}('v) . ('b{0} < 'u{0} > | 'b{0} < 'w{0} >)]

```



```

eq TQ = [[{'a{0}' 'b{0}' 'w{0}']
           'b{0}('u) . ('b{0} < 'u{0} > | 'b{0} < 'w{0} >)]

```

The metarepresentation of these `TTrms` can now be obtained by using `'TP.TTrm` and `'TQ.TTrm`, and we can then check for the may-testing preorder between the given processes as follows:

```

Maude> red 'TP.TTrm maypre 'TQ.TTrm .
reduce in APITRACESET : 'TP.TTrm maypre 'TQ.TTrm .
rewrites: 791690 in 2140ms cpu (2160ms real) (361422 rewrites/second)
result Bool: true
Maude> red 'TQ.TTrm maypre 'TP.TTrm .
reduce in APITRACESET : 'TQ.TTrm maypre 'TP.TTrm .
rewrites: 664833 in 1620ms cpu (1640ms real) (410390 rewrites/second)
result Bool: false

```

Thus, we have $P \sqsubseteq Q$, but $Q \not\sqsubseteq P$. The reader can check that indeed, $\|Q\|_{fn(P,Q)} \lesssim \|P\|_{fn(P,Q)}$, but $\|P\|_{fn(P,Q)} \not\lesssim \|Q\|_{fn(P,Q)}$.

7 Conclusions and Future Work

In this paper, we have described an executable specification in Maude of the operational semantics of an asynchronous version of the π -calculus using conditional rewrite rules with rewrites in the conditions as proposed by Verdejo and Martí-Oliet in [18], and the CINNI calculus proposed by Stehr in [14] for managing names and their binding. In addition, we also implemented the may-testing preorder for π -calculus processes using the Maude metalevel, where we use the `metaSearch` operation to calculate the set of all traces for a process and then compare two sets of traces according to a preorder relation between traces. As emphasized throughout the paper, the new features introduced in Maude 2.0 have been essential for the development of this executable specification, including rewrites in conditions, the `frozen` attribute, and the `metaSearch` operation.

An interesting direction of further work is to extend our implementation to the various typed variants of π -calculus. Two specific typed asynchronous π -calculi for which the work is under way are the local π -calculus ($L\pi$) [10] and the Actor model [1,15]. Both of these formal systems have been used extensively in formal specification and analysis of concurrent object-oriented languages [2,8], and open distributed and mobile systems [9]. The alternate characterization of may testing for both of these typed calculi was recently published [16,17]. We are extending the work presented here to account for the type systems for these calculi, and modifications to the trace based characterization of may testing. We are also looking for interesting concrete applications to which this can be applied; such experiments may require extending our implementation to extensions of π -calculus with higher level constructs, although these may just be syntactic sugar.

Acknowledgements

This research has been supported in part by the Defense Advanced Research Projects Agency (contract numbers F30602-00-2-0586 and F33615-01-C-1907), the ONR MURI Project *A Logical Framework for Adaptive System Interoperability*, and the Spanish CICYT project *Desarrollo Formal de Sistemas Basados en Agentes Móviles* (TIC2000-0701-C02-01). This work was done while the last author was visiting the Department of Computer Science in the University of Illinois at Urbana-Champaign, for whose hospitality he is very grateful. We would like to thank José Meseguer for encouraging us to put together several complementary lines of work in order to get the results described in this paper.

References

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

- [2] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.
- [3] M. Boreale and R. De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120:279–303, 1995.
- [4] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. In *Proceedings 14th IEEE Symposium on Logic in Computer Science, LICS’99, Trento, Italy, July 2–5, 1999*, pages 157–166. IEEE Computer Society Press, 1999.
- [5] M. Boreale, R. De Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes. *Information and Computation*, 172(2):139–164, 2002.
- [6] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA’96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 35–50. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>.
- [7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 297–318. Elsevier, 2000. <http://www.elsevier.nl/locate/entcs/volume36.html>.
- [8] I. A. Mason and C. Talcott. A semantically sound actor translation. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP’97, Bologna, Italy, July 7–11, 1997, Proceedings*, volume 1256 of *Lecture Notes in Computer Science*, pages 369–378. Springer-Verlag, 1997.
- [9] M. Merro, J. Kleist, and U. Nestmann. Local π -calculus at work: Mobile objects as mobile processes. In J. van Leeuwen et al., editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000 Sendai, Japan, August 17–19, 2000, Proceedings*, volume 1872 of *Lecture Notes in Computer Science*, pages 390–408. Springer-Verlag, 2000.
- [10] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In K. G. Larsen, S. Skjum, and G. Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP’98, Aalborg, Denmark, July 13–17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 856–867. Springer-Verlag, 1998.
- [11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.
- [12] R. De Nicola and M. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.

- [13] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, September 1981.
- [14] M.-O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ -, ς - and π -calculi. In K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71–92. Elsevier, 2000. <http://www.elsevier.nl/locate/entcs/volume36.html>.
- [15] C. Talcott. An actor rewriting theory. In J. Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 360–383. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>.
- [16] P. Thati, R. Ziaeи, and G. Agha. A theory of may testing for actors. In B. Jacobs and A. Rensink, editors, *Proceedings IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002), March 20–22, 2002, Enschede, The Netherlands*, pages 147–162. Kluwer Academic Publishers, 2002.
- [17] P. Thati, R. Ziaeи, and G. Agha. A theory of may testing for asynchronous calculi with locality and no name matching. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9–13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
- [18] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In U. Montanari, editor, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. (This volume.) <http://www.elsevier.nl/locate/entcs/volume71.html>.
- [19] P. Viry. Input/output for ELAN. In J. Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 51–64. Elsevier, 1996. <http://www.elsevier.nl/locate/entcs/volume4.html>.

A Appendix

The diagram in Figure A.1 illustrates the graph of module importation in our implementation that closely follows the structure of the paper. The complete code is available at <http://osl.cs.uiuc.edu/~ksen/api/>. Here we only show the module that contains the rewrite rules for the operational semantics

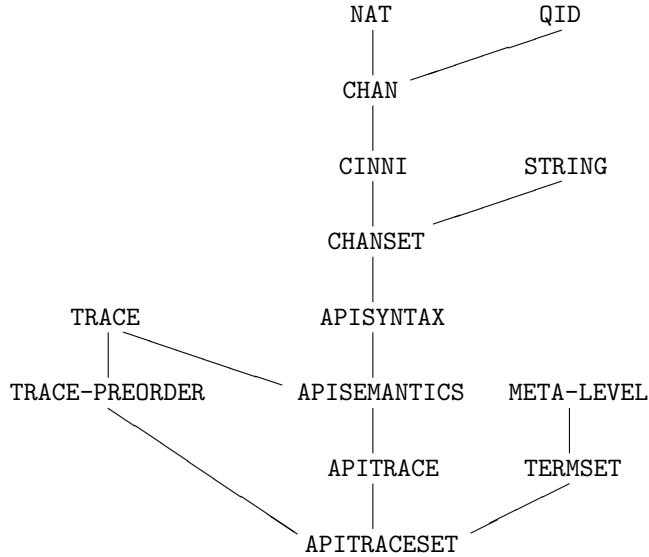


Fig. A.1. The graph of module importation in the implementation.

of asynchronous π -calculus (Table 2). The function `genQid` used in the condition of the last `Res` rule generates an identifier that is fresh, i.e. an identifier not used to construct channel names in the set passed as the argument to the function.

```

mod APISEMANTICS is
  inc APISYNTAX .
  inc CHANSET .
  inc TRACE .
  sorts EnvTrm TraceTrm .
  subsort EnvTrm < TraceTrm .

  op  []_ : Chanset Trm -> EnvTrm [frozen] .
  op  {}_ : Action TraceTrm -> TraceTrm [frozen] .
  op  notinfn : Qid Trm -> Prop .

  vars N : Nat .           vars X Y Z : Qid .
  vars CX CY : Chan .     var CS CS1 CS2 : Chanset .
  vars A : Action .        vars P1 Q1 P Q : Trm .
  var SUM : SumTrm .       var IO : ActionType .

  eq  notinfn(X,P) = not X{0} in freenames(P) .

  rl [Inp] : [CY CS] (CX(X) . P) =>
             {f(i,CX,CY)} ([CY CS] ([X := CY] P)) .

  rl [Inp] : [CY CS] ((CX(X) . P) + SUM) =>
             {f(i,CX,CY)} ([CY CS] ([X := CY] P)) .

  rl [Tau] : [CS] (tau . P) => { tauAct } ([CS] P) .

  rl [Tau] : [CS] ((tau . P) + SUM) => { tauAct } ([CS] P) .
  
```

```

crl [BInp] : [CS] P => {b(i,CX,'u)} ['u{0} [shiftup 'u] CS] P1
    if (not flag in CS) /\
        CS1 := flag 'u{0} [shiftup 'u] CS /\
            [CS1] [shiftup 'u] P => {f(i,CX,'u{0})} [CS1] P1 .

r1 [Out] : [CS] CX < CY > => { f(o,CX,CY) } ([CS] nil) .

crl [Par] : [CS] (P | Q) => {f(IO,CX,CY)} ([CS] (P1 | Q))
    if [CS] P => {f(IO,CX,CY)} ([CS] P1) .

crl [Par] : [CS] (P | Q) =>
    {b(IO,CX,Y)} [Y{0} ([shiftup Y] CS)] (P1 | [shiftup Y] Q)
    if [CS] P => {b(IO,CX,Y)} ([CS1] P1) .

crl [Com] : [CS] (P | Q) => {tauAct} ([CS] (P1 | Q1))
    if [CS] P => {f(o,CX,CY)} ([CS] P1) /\
        [CY CS] Q => {f(i,CX,CY)} ([CY CS] Q1) .

crl [Close] : [CS] (P | Q) => {tauAct} [CS] new [Y] (P1 | Q1)
    if [CS] P => {b(o,CX,Y)} [CS1] P1 /\
        [Y{0} [shiftup Y] CS] [shiftup Y] Q =>
            {f(i,CX,Y{0})} [CS2] Q1 .

crl [Res] : [CS] (new [X] P) =>
    {[shiftdown X] f(IO,CX,CY)} [CS] (new [X] P1)
    if CS1 := [shiftup X] CS /\
        [CS1] P => {f(IO,CX,CY)} [CS1] P1 /\
        (not X{0} in (CX CY)) .

crl [Res] : [CS] (new [X] P) => {tauAct} [CS] (new [X] P1)
    if [CS] P => {tauAct} [CS] P1 .

crl [Res] : [CS] (new [X] P) =>
    {[shiftdown X] b(o,CX,Z)} [Z{0} CS] new[X]([ Y := Z{0} ] P1)
    if Z := genQid(X{0} CS freenames(P)) /\
        {[shiftup X] CS] P => {b(o,CX,Y)} [CS1] P1 /\
        X{0} /= CX .

crl [Open] : [CS] (new[X] P) => {[shiftdown X] b(o,CY,X)} [X{0} CS1] P1
    if CS1 := [shiftup X] CS /\
        [CS1] P => {f(o,CY,X{0})} [CS1] P1 /\
        X{0} /= CY .

crl [If] : [CS1] (if CX = CX then P else Q fi) => {A} [CS2] P1
    if [CS1] P => {A} [CS2] P1 .

crl [Else] : [CS1] (if CX = CY then P else Q fi) => {A} [CS2] Q1
    if CX /= CY /\
        [CS1] Q => {A} [CS2] Q1 .

crl [Rep] : [CS1] (! P) => {A} [CS2] P1
    if [CS1] (P | (! P)) => {A} [CS2] P1 .

endm

```

13

Thin Middleware for Ubiquitous Computing

**Koushik Sen
Gul Agha**

13.1 INTRODUCTION

As Donald Norman put it in his popular book *Invisible Computers* [12], “a good technology is a disappearing technology.” A good technology seamlessly permeates into our lives in such a way that we use it without noticing its presence. It is invisible until it is not available. Ever since the invention of microprocessors, many computer researchers have strived to make computer technology a “good” technology.

Advances in chip fabrication technology have reached a point where we can physically make computing devices disappear. Bulkier machines have given way to smaller yet more powerful personal computers. It has become possible to implant a complete package of a microprocessor with wireless communication, storage, and a sensor on a cubic millimeter silicon die [8]. Specialized printers print out computer chips on a piece of plastic paper [6]. Computer chips are woven into on a piece of fabric [9]. “Smart labels” (a.k.a passive RFID tags) [7] will soon be attached to every product in the market.

The growth of devices with embedded computers will provide task-oriented, simple services which are highly optimized for their operating environment. More user oriented, human friendly services may be created by networking the embedded nodes, and coordinating their software services.

Composing existing component services to create higher-level services has been promoted by CORBA, DCOM, Jini, and similar middleware platforms. However, these middleware services were designed without paying much attention to resource management issues pertinent to embedded nodes: the middlewares tend to have large

footprints that do not fit into the typically small memories of tiny embedded computers. Thus there is a need for a middleware which allows components to be glued together without a large overhead.

Embedded nodes are autonomous and self contained. They have their own state and a single thread of control, and a well defined interface for interaction. Interaction between nodes is asynchronous in nature. The operating environment for these devices may be unreliable. For these reasons, the interaction between different devices must be arms-length – the failure of one device should not affect another. For example, consider an intelligent home where the clock is networked to the coffee maker and an alarm also triggers the coffee maker. An incorrectly operating coffee maker should not cause the alarm clock to fail to operate. The autonomy and asynchrony in the model we describe helps ensure such fault containment.

Embedded nodes are typically resource constrained – they have small communication range, far less storage, limited power supply, etc. These resource limitations have a number of consequences. Small memory means that not every piece of code that may be required over the lifetime of a node can be pre-loaded onto the node. Limited power supply means that certain abstractions, such as those requiring busy waiting to implement, may be too expensive to be practical.

We propose *thin middleware* as a model for network-centric, resource-constrained embedded systems. There are two aspects to the thin middleware model. First, we represent component services as *Actors* [1, 3]. The need for autonomy and asynchrony in resource-constrained networks of embedded nodes makes Actors an appropriate model to abstractly represent services provided by such systems. Service interactions in the thin middleware are modeled as asynchronous communications between actors. Second, we introduce the notion of meta-actors for service composition and customization. Meta-actors represent system level behavior and interact with actors using an event-based signal/notify model.

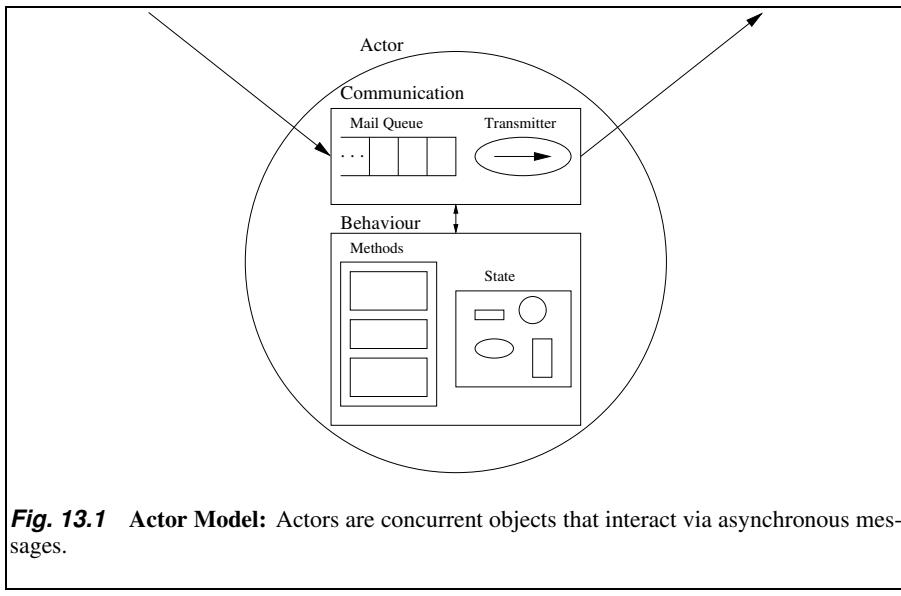
13.2 ACTORS

Actors [1, 3] were developed as a basis for modeling distributed systems. An actor encapsulates a state, a set of procedures which manipulate the state, and a thread of control. Each actor has a unique *mail address* and a *mail buffer* to receive messages. Actors compute by serially processing messages queued in their mail buffers. An actor waits if its mail buffer is empty. Actors interact by sending messages to each other.

In response to a message, an actor carries out a local computation (which may be represented by any computer program) and three basic kinds of actions (see Figure 13.1):

Send messages: an actor may *send* messages to other actors. Communication is point-to-point and is assumed to be weakly fair: executing a *send* eventually causes the message to be buffered in the mail queue of the recipient. Moreover, messages are by default asynchronous and may arrive in an order different from the one in which they were sent.

Create actors: An actor may *create* new actors with specified behaviors. Initially, only the creating actor knows the name of the new actor. However, actor names are first class entities which may be communicated in messages; this means



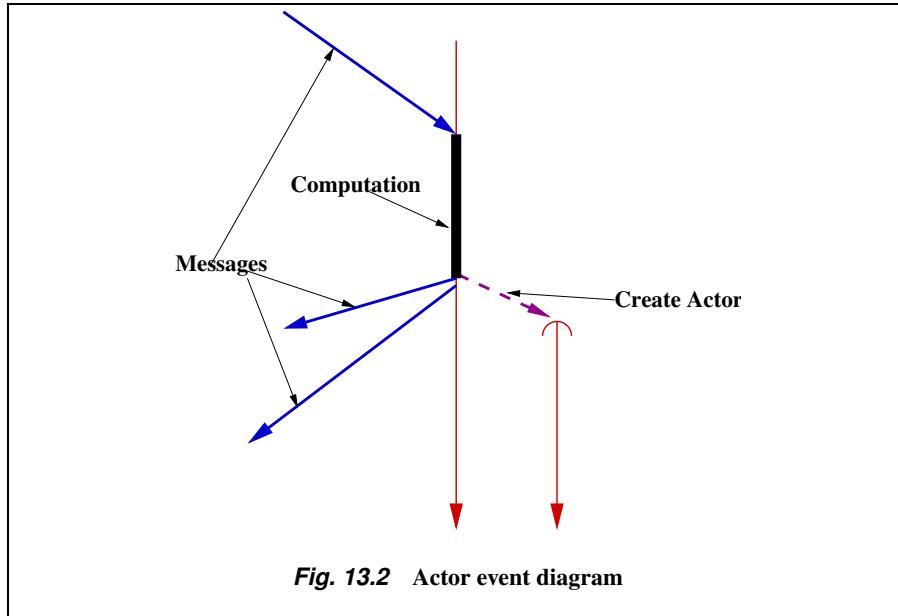
that coordination patterns between actors may be dynamic and the system is extensible.

Become ready to accept a message: The actor becomes *ready* to process the next message in its mail queue. If there is no message in its mail queue, the actor waits until a new message arrives and processes it.

Asynchronous message passing is the distributed analog of method invocation in sequential object-oriented languages. The *send* and *create* operations can be thought of as explicit requests, while the *ready* operation is implicit at the end of a method. That is, actors do not explicitly indicate that they are ready to receive the next message. Rather, the system automatically invokes *ready* when an actor method completes.

Actor computations are abstractly represented using *actor event diagrams* as illustrated in Figure 13.2. Two kinds of objects are represented in such diagrams: actors and messages. An actor is identified with a vertical line which represents the life-line of the actor. The darker parts on the line represent the processing of a message by the actor. The actor may create new actors (dotted lines) and may send messages (solid lines) to other actors. The messages arrive at their target actors after arbitrary but finite delay and get enqueued at the target actor's mail queue.

Note that the nondeterminism in actor systems results from possible shuffles of the order in which messages are processed. There are two causes of this nondeterminism. First, the time taken by a message to reach the target actor depends on factors such as the route taken by the message, network traffic load, and the fault-tolerance protocols used. Second, the order in which messages are sent may itself be affected by the processing speed at a node and the scheduling of actors on a given node. Nondeterminism in the order of processing messages abstracts over possible communication and scheduling delays.

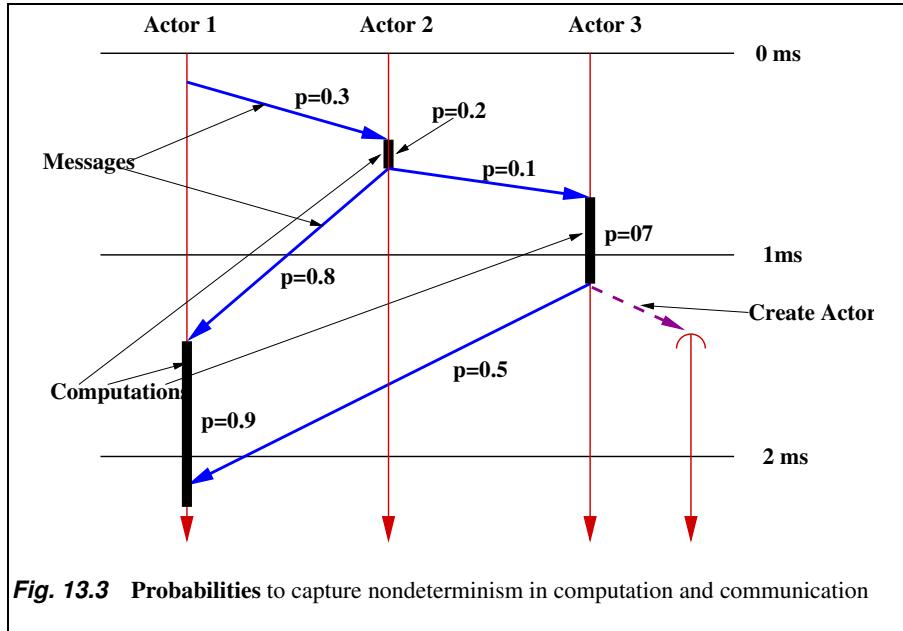


The nondeterministic model of concurrency provides a loose specification. Properties expressed in this model state what *may*, or what *must*, eventually happen. In reality, the probability that, for example, a message sent at a given time will be received after a million years is practically infinitesimal. One way to express constraints on the arbitrary interleavings resulting from a purely nondeterministic model is by using a probabilistic model. In such a model, we associate a probability with each transition which may depend on the current state of the system. Our specifications then say something about when something may happen with a given probability. We discuss a probabilistic model in some more detail below.

13.2.1 Probabilistic Discrete Real-time Model

Traditional models of concurrent computation do not assume a unique global clock – rather each actor is asynchronous (for example, see [3, 2]). However, when modeling interaction of the physical world with distributed computation, it is essential to consider guarantees in real-time. Such guarantees are expressed in terms of a unique global time or wall clock and the behavior of all devices and nodes is modeled in terms of this reference time (for example, see [11, 14, 13, 10]). This amounts to a synchronous model of actors and it implies a ‘tight coupling’ in the implementation of actors; network and scheduling delays, as well as clock drift on the nodes, must be severely restricted.

In network embedded systems, a number of factors make a tight coupling in the implementation of actors infeasible. For example, the operation of some embedded devices may be unreliable, and message delivery may have nondeterministic delays due to transmission failures, collisions, and message loss. So in large network embed-



ded systems, it is not feasible to maintain a unique reference clock. The introduction of probability in the operations can be thought of as an intermediate synchronization model. In a probabilistic model, we assume that the embedded nodes agree on a global clock, but their drift from the clock is only probabilistically bound. Such probabilities replace the qualitative nondeterminism in computation and communication.

We assume the actors and the messages in transit form a *soup*. The components of the system follow a reference clock with some probability. The global time of the whole system (soup) advances in discrete time steps. The time steps can be compressed and stretched, depending on the kind of property we want to express. For example, the time step can be set to one second or it may be set to one millisecond. The global time of the system advances by one step when all the *actions* (computation and communication) that are possible in that time step have happened (see Figure 13.3). We associate a local clock with each actor and it advances with every global time step. However, it is reset to zero when the actor consumes a message. The clock remains zero when the actor is idle.

At a given time step an actor may be in one of three states:

- ready to process a message from its mail queue,
- busy computing, or
- waiting, because there is no message in mail queue.

If the actor is in either of the first two states, it can take the following actions:

- *complete* the computation in its current time step; or,
- *delay* its computation by one time step.

In the first case, the local clock of the actor remains same and so it is open to other actions in that time step. However, for the second action the local clock of the actor advances by one time step and hence, all possible actions of that actor get disabled for that time step. The two actions get enabled once the global time advances to the next time step. As a function of the state of the system, we associate different probabilities with each of the two actions.

Similarly, at a given time step a message can take three actions:

- it can get *enqueued* at the target actor,
- it can get *lost* and thus removed from the soup, or
- it can get *delayed*, in transit, by one time step.

If a message is delayed in transit, the local time of the message advances by one time step and so the message cannot take any more actions in that global time step. However, all the three actions will get enabled once the global time advances to the next step. Probabilities are associated with each action. The probabilities depend on factors such as the message density in the route taken by the message, the time for which it has been delayed (value of local clock of the message), and the number of messages sharing the same communication channel.

A computation path is defined as a sequence of states that the system has seen in the course of its computation. Note that the system retains the same computation paths as it would have in a nondeterministic model of concurrency. The probability that a particular finite sequence of states in a path will occur is obtained by multiplying the probabilities of all the actions in that sequence of states. Some of these probabilities will grow sufficiently small that they will no longer be relevant to the proof of some properties of our interest.

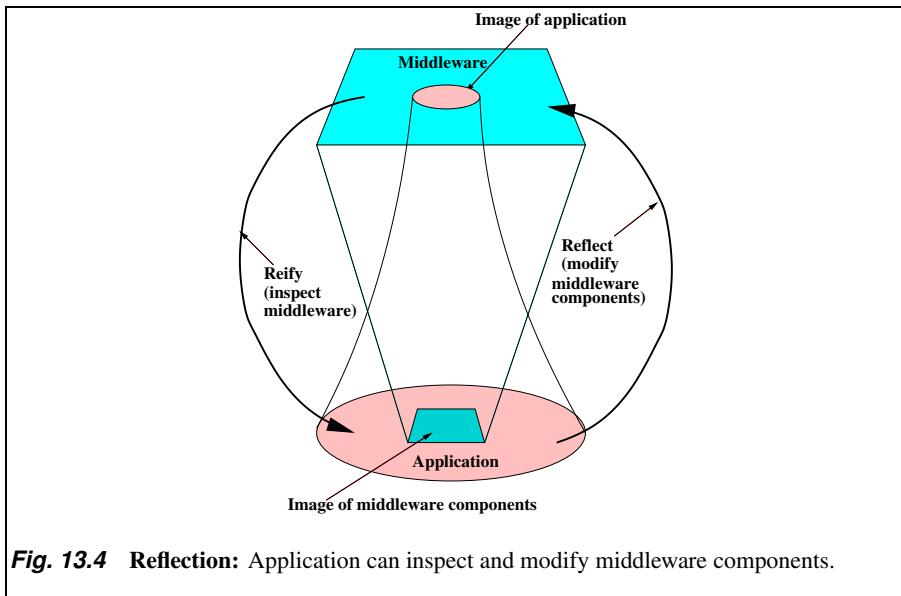
Using the above model, we can express properties of the form: “Within time t , the system will reach a state which satisfies a property π with probability p .” For example:

- the alarm clock will ring at 7:00 a.m. with probability 0.99.
- the microwave will complete popping 95% of the popcorn by 10 a.m. with probability 0.98.

In implementing probabilistic timing specifications, one constrains the system level behavior which involves networks of heterogeneous nodes. A middleware provides a uniform interface to access such nodes. We represent the middleware itself as a collection of actors. The model we describe enables dynamic customizability of the execution environment of an actor in order to satisfy properties such as timing and security.

13.3 REFLECTIVE MIDDLEWARE

A key requirement for middleware is that it must enable dynamic customization – so that services can be pushed in and pulled out at runtime. This scheme of pushing-in and pulling-out of services allows the middleware to keep on a node only those services that are required by an application. The result is a light weight middleware.



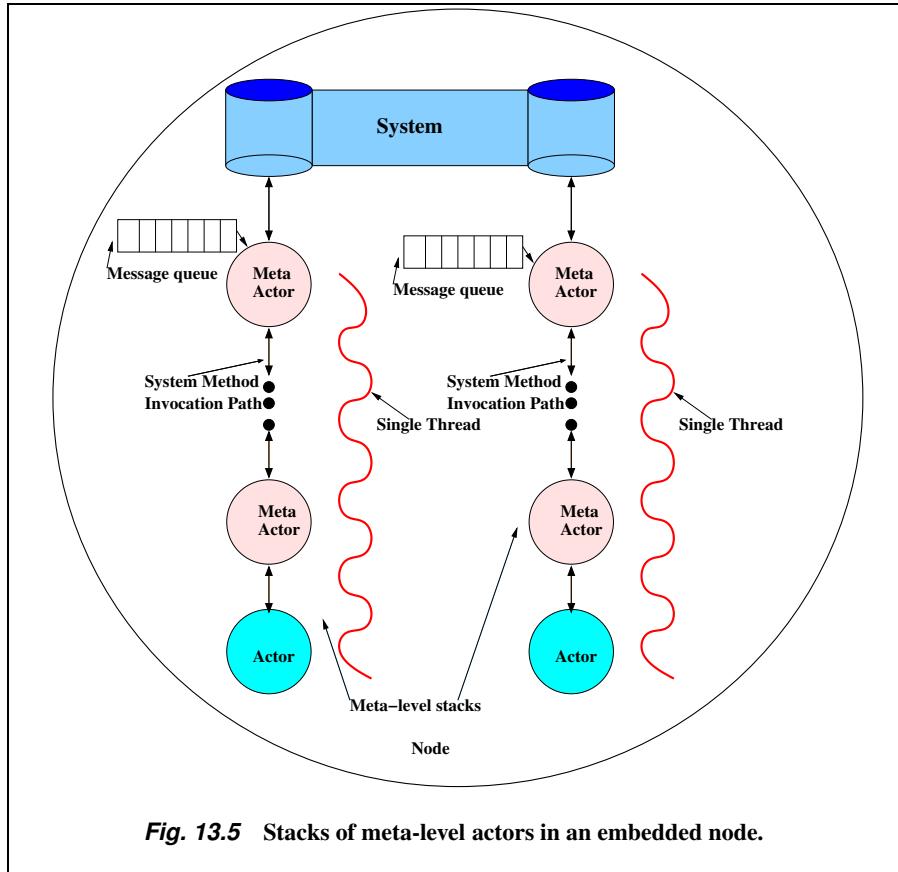
Because an application may be aware of system level requirements for timing, security, or messaging protocols, it needs to have access to the underlying system. We support the ability of an application to modify its system level requirements by dynamically changing the middleware through the use of *computational reflection*.

A reflective middleware provides a representation of its different components to the applications running on top of it. The applications can *inspect* this representation and modify it. The modifications made to the components are immediately mirrored to the application. In this way, applications can dynamically customize the different components of the middleware through reflection (see Figure 13.4).

We use the *meta-actor* extension of actors to provide a mechanism of architectural customization [5]. A system is composed of two kinds of actors: base actors and meta-actors. Base actors carry out application-level computation, while meta-level actors are part of the runtime system (middleware) that manages system resources and controls the base-actor's runtime semantics.

13.3.1 Meta-architecture

From a systems point of view, actors do not directly interact with each other: instead, actors make *system method* calls which request the middleware to perform a particular action. A system method call which implements an actor operation is always 'blocking': the actor waits till the system signals that the operation is complete. Middleware components which handle system method calls are called *meta-actors*. A meta-actor executes a method invoked by another actor and returns on the completion of the execution. The requisite synchronization between an actor and its meta-actor is facilitated by treating the meta-actor as a passive object: it does not have its own thread of control. Instead, the calling object is suspended. In other words, an actor and its



meta-actor are not concurrent – the latter represents the system level interpretation of the behavior of the former.

A meta-actor is capable of customizing the behavior of another actor by executing the method invoked by it. An actor customized in this fashion is referred to as the *base actor* relative to its meta-actor. To provide the most primitive model of customization a meta-actor can customize a single base-actor. However, multiple customizations may be applied to a single actor by building a *meta-level* stack, where a meta-level stack consists of a single actor and a stack of meta-actors (see Figure 13.5). Each meta-actor customizes the actor which is just below it in the stack. Messages received by an actor in a meta-level stack are always delegated to the top of the stack so that the meta-actor always controls the delivery of messages to its base-actor. Similarly messages sent by an actor pass through all the meta-actors in the stack.

We identify each operation of a base-actor as a system method call as follows.

- **send(msg):** This operation invokes the system method `transmit` with `msg` (`msg` is the message sent by the actor) as argument. If the actor has a meta-actor on its top it calls the `transmit` method of the meta-actor and wait for its

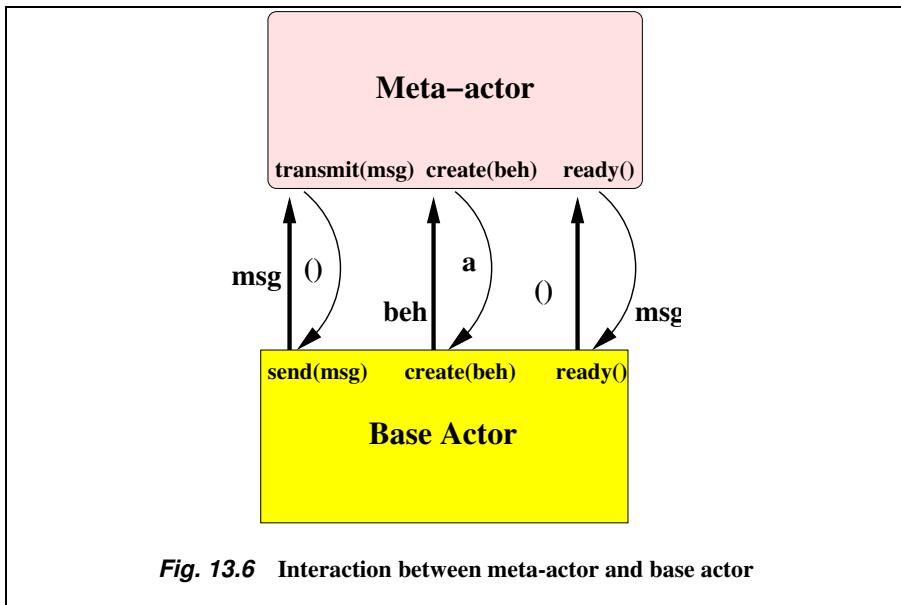


Fig. 13.6 Interaction between meta-actor and base actor

return. The method returns without any value. Otherwise, if the actor is not customized by a meta-actor, it passes the message to the system for sending.

- **create(beh):** This operation invokes the system method `create` with the given `beh` (`beh` is the behavior with which the newly created actor will be instantiated) as argument. If there is a meta-actor on top of the actor, it calls the `create` method of the meta-actor and waits for its return. The method returns the address `a` of the new actor. Otherwise, the actor passes the `create` request to the system.
- **ready():** The system method `ready` is invoked when an actor has completed processing the current message and is waiting for another message. If the actor has a meta-actor on top it calls the `ready` method of the meta-actor and waits for its return. The method returns a message to the base-actor. Otherwise, the actor picks up a message from its mail queue and processes it. Notice, there is a single mail-queue for a given meta-level stack.

The method call-return mechanism for different actor operations and the availability of a single queue for a meta-level stack makes the execution of a meta-level stack single threaded. So explicit scheduling of each actor in the stack is not required. The meta-actors behave as reactive passive objects which respond only when a system method is invoked by its base actor. The single thread implementation of a meta-level stack is important, as most of the embedded devices can have a single thread only. An example of such an embedded OS is TinyOS which runs on motes.

Every meta-actor has a default implementation of the three system methods. These implementations may be described as follows:

- **transmit(*msg*):** If there is a meta-actor on its top, it calls `transmit (msg)` method of that meta-actor and waits for it to return. Otherwise, it asks the system to send the message to the target and returns.
- **create(*beh*):** If the actor has a meta-actor at its top, it calls `create (beh)` method of that meta-actor and waits for the actor to return with an actor address. Otherwise, the actor passes the create request to the system and waits till it gets an actor address from the system. After receiving new actor address, the actor returns it to the base actor.
- **ready():** If there is a meta-actor on top of it, it calls `ready ()` method of that meta-actor and waits for it to return a message. Otherwise, the actor, by definition located at the top of the meta-level stack, dequeues a message from the mail queue. After getting the message, the actor returns the message to the base actor.

```

actor Encrypt(actor receiver) {
    // Encrypt outgoing
    // messages if they
    // are targeted to
    // the receiver
    method transmit(Msg msg) {
        actor target = msg.dest;
        if (target == receiver)
            target ← encrypt(msg);
        else
            target ← msg;
        return;
    }
}

actor Decrypt() {
    // Decrypt incoming messages
    // targeted for
    // base actor (if necessary)
    method ready() {
        Msg msg = ready();
        if (encrypted(msg))
            return(decrypt(msg));
        else
            return(msg);
    }
}

```

Fig. 13.7 Meta-Level Implementation of Encryption: The `Encrypt` meta-actor intercepts `transmit` signals and encrypts outgoing messages. The `Decrypt` policy actor intercepts messages targeted for the receiver (via the `rcv` method) and, if necessary, decrypts an incoming message before delivering it.

As an example of how we may customize actors under this model, consider the encryption of messages between a pair of actors. Figure 13.7 gives pseudo-code for a pair of meta-actors which may be installed at each endpoint. The `Encrypt` meta-actor implements the `transmit` method which is called by the base-actor while sending a message. Within `transmit`, a message is encrypted before it is sent to its target. The `Decrypt` meta-actor implements the `ready` method which is called when the base actor is ready to process a message. Method `ready` decrypts the message before returning the message to the base-actor.

The abstraction of the middleware in terms of meta-actors gives the power of dynamic customization. Meta-actors can be installed or pulled out dynamically. This pushing in and pulling out of meta-actors by the application itself makes it capable of customizing the middleware. It also makes it possible to have only those middleware components which are required by services of the current application – facilitating our goal of thin middleware.

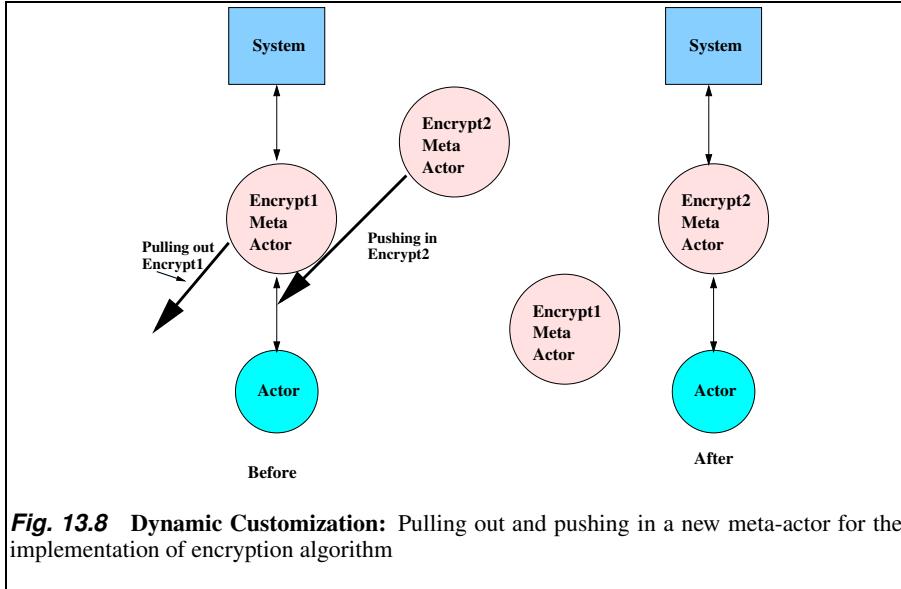


Fig. 13.8 Dynamic Customization: Pulling out and pushing in a new meta-actor for the implementation of encryption algorithm

13.4 DISCUSSION

We have described some preliminary work on a model of reflective middleware. We believe that further development of thin middleware will be central to the future integration of computing and the physical world [4]. However, many important problems have to be addressed before such an integration can be realized. We describe two areas to illustrate the problems. These areas relate, respectively, to the model and implementation of middleware.

A formal model of the interaction of the properties of actors and meta-actors has been developed in terms of a *two-level semantics* [15]. This model needs to be extended to its probabilistic real-time counterpart. For example, methods for composition of transition probabilities for actors and their meta-actors have not been developed.

More research is required in the implementation of thin middleware. Current implementation of reflective actor middleware has been based on high-level languages – which necessarily assume a large infrastructure. An alternate implementation would be in terms of a very efficient and small virtual machine which allows enforcement of timing properties. Related problems are incrementally compiling high-level code to such a virtual machine and supporting the mobility of actors executing on the virtual machine.

In our view, the solution to these and related problems define an ambitious research agenda for the coming decade.

13.5 ACKNOWLEDGMENTS

The research described here has been supported in part by the Defense Advanced Research Projects Agency (Contract numbers: F30602-00-2-0586 and F33615-01-C-1907). We would like to thank Nadeem Jamali and Nirman Kumar for reviewing previous versions of this paper and giving feedback.

REFERENCES

1. G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.
2. G. Agha. Modeling Concurrent Systems: Actors, Nets, and the Problem of Abstraction and Composition. In *17th International Conference on Application and Theory of Petri Nets*, Osaka, Japan, June 1996.
3. G. Agha, I. A. Mason., S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
4. Gul A. Agha. Adaptive middleware. *Communications of the ACM*, 45(6):30–32, June 2002.
5. M. Astley and G. Agha. Customization and composition of distributed objects: Middleware abstractions for policy management. In *Proceedings of the Sixth International Symposium of Foundations of Software Engineering*, pages 1–9, 1998.
6. S. B. Fuller, E. J. Wilhelm, and J. M. Jacobson. Ink-jet printed nanoparticle microelectromechanical systems. *Journal of Microelectromechanical Systems*, 11(1):54–60, 2002. <http://www.media.mit.edu/molecular/projects.html>.
7. AIM. Inc. <http://www.aimglobal.org/technologies/rfid/>.
8. J. M. Kahn, R. H. Katz, and K. S. J. Pister. Mobile Networking for Smart Dust. In *ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99)*, Seattle, WA, August 1999. <http://robotics.eecs.berkeley.edu/pister/SmartDust/>.
9. MIT Media Lab. <http://lcs.www.media.mit.edu/projects/wearables/>.
10. B. Nielsen and G. Agha. Semantics for an Actor-Based Real-Time Language. In *4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*. Submitted. Naval Surface Warfare Center Dahlgren Division/IEEE, April 1995. In conjunction with 10th IEEE Int. Parallel Processing Symposium (IPPS), Honolulu, Hawaii, USA.
11. B. Nielsen and G. Agha. Towards reusable real-time objects. *Annals of Software Engineering: Special Volume on Real-Time Software Engineering*, 7:257–282, 1999.
12. D. Norman. *The Invisible Computer*. The MIT Press, Cambridge, MA, USA, 1998.

13. S. Ren. *An Actor-Based Framework for Real-Time Coordination*. PhD thesis, Department Computer Science. University of Illinois at Urbana-Champaign, 1997. PhD. Thesis.
14. S. Ren, G. Agha, and M. Saito. A modular approach for programming distributed real-time systems. *Journal of Parallel and Distributed Computing*, 36(1):4–12, 1996. Also published in *School on Embedded Systems, European Educational Forum 1996*, pp 52–72.
15. Nalini Venkatasubramanian and Carolyn L. Talcott. Reasoning about meta level activities in open distributed systems. In *Symposium on Principles of Distributed Computing*, pages 144–152, 1995.

Author(s) affiliation:

• **Koushik Sen, and Gul Agha**

Department of Computer Science
University of Illinois at Urbana-Champaign
Email: [ksen,agha]@uiuc.edu